



Intel® Integrated Performance Primitives for Intel® Architecture

Reference Manual

Volume 1: Signal Processing

Document Number: A24968-017US

World Wide Web: <http://developer.intel.com>

Version	Version Information	Date
-1001	Original issue.	09/00
-1002	Documents Intel IPP 1.0 final release. Functions NormDiff, AutoCorr, and ZeroMean have been added. Derivatives Functions section have been revised.	02/01
-1101	Documents Intel IPP 1.1 beta release.	04/01
-2001	Documents Intel IPP 2.0 beta release. General audio coding, MP3, and transcendental vector functions have been added. Speech recognition API have been revised.	08/01
-2002	Documents Intel IPP 2.0 release. New Intel IPP common functions have been added. The set of arithmetic, vector initialization, statistical, and filtering functions have been expanded.	11/01
-3001	Documents Intel IPP 3.0 pre-beta. New speech codec functions have been added. The set of conversion and windowing functions have been expanded. Speech recognition API have been updated.	04/02
-3002	Documents Intel IPP 3.0 beta.	06/02
-3003	Documents Intel IPP 3.0 beta update.	09/02
-3004	Documents Intel IPP 3.0 release.	11/02
-4001	Documents Intel IPP 4.0 beta. New functions for cross-architecture development have been added.	05/03
-4002	Documents Intel IPP 4.0 release.	10/03
-012	Documents Intel IPP 4.1 beta release.	04/2004
-013	Documents Intel IPP 4.1 release. Added new statistical functions and flavors for arithmetic and filtering functions.	07/2004
-014	Documents Intel IPP 5.0 beta release. New data compression functions and functions for regular expressions have been added. The set of arithmetic, conversion, transform, and filtering functions have been expanded.	03/2005
-015	Documents Intel 5.0 release. Added new data compression functions and flavors for vector initialization and Fourier transform functions.	08/2005
-016	Documents Intel IPP 5.1 beta release.	10/2005
-017	Documents Intel IPP 5.1 release. Added new flavors for arithmetic functions.	02/2006

The information in this manual is subject to change without notice and Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. The information in this document is provided in connection with Intel products and should not be construed as a commitment by Intel Corporation.

EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

Intel, the Intel logo, Intel SpeedStep, Intel NetBurst, Intel NetStructure, MMX, Intel386, Intel486, Celeron, Intel Centrino, Intel Xeon, Intel XScale, Itanium, Pentium, Pentium II Xeon, Pentium III Xeon, Pentium M, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2000 - 2006, Intel Corporation.

Contents

Chapter 1 Overview

About This Software	1-1
Hardware and Software Requirements	1-2
Platforms Supported	1-2
Cross-Architecture Alignment	1-2
Cross Architecture Overview	1-2
API Changes in Version 5.0	1-3
Technical Support	1-4
Intel IPP Code Samples	1-5
About This Manual	1-5
Manual Organization	1-5
Function Descriptions	1-6
Audience for This Manual	1-7
Online Version	1-7
Related Publications	1-7
Notational Conventions	1-7
Font Conventions	1-7
Signal Name Conventions	1-8
Naming Conventions	1-8

Chapter 2 Intel® Integrated Performance Primitives Concepts

Basic Features	2-1
Function Naming	2-2

Data-Domain.....	2-2
Name	2-2
Data Types	2-3
Descriptor	2-5
Arguments	2-6
Structures and Enumerators.....	2-6
Library Version Structure	2-6
Complex Data Structures.....	2-7
Function Context Structures	2-7
Enumerators	2-7
Data Ranges.....	2-11
Data Alignment	2-11
Integer Scaling.....	2-11
Error Reporting	2-12
Code Examples	2-17

Chapter 3 **Support Functions**

Version Information Functions	3-2
ippGetLibVersion.....	3-2
Memory Allocation Functions	3-4
ippMalloc.....	3-4
ippFree	3-5
Common Functions	3-6
ippGetStatusString	3-6
ippGetCpuType	3-7
ippGetCpuClocks	3-8
ippGetCpuFreqMhz	3-9
ippGetMaxCacheSizeB	3-9
ippSetFlushToZero	3-10
ippSetDenormAreZeros	3-11
ippAlignPtr	3-12
ippSetNumThreads	3-13
ippGetNumThreads	3-13
ippMalloc	3-14

ippFree	3-15
Dispatcher Control Functions	3-16
ippStaticInit	3-16
ippStaticInitCpu	3-17

Chapter 4

Vector Initialization Functions

Vector Initialization Functions	4-3
Copy	4-3
Move	4-5
Set	4-6
Zero	4-7
Sample-Generating Functions	4-8
Tone-Generating Functions	4-9
ToneInitAllocQ15	4-9
ToneFree	4-10
ToneGetStateSizeQ15	4-11
ToneInitQ15	4-12
ToneQ15	4-13
Tone_Direct	4-14
ToneQ15_Direct	4-15
Triangle-Generating Functions	4-17
TriangleInitAllocQ15	4-18
TriangleFree	4-19
TriangleGetStateSizeQ15	4-20
TriangleInitQ15	4-21
TriangleQ15	4-22
Triangle_Direct	4-23
TriangleQ15_Direct	4-25
Uniform Distribution Functions	4-26
RandUniformInitAlloc	4-26
RandUniformFree	4-27
RandUniformInit	4-28
RandUniformGetSize	4-29

RandUnifrom	4-29
RandUniform_Direct	4-30
Gaussian Distribution Functions	4-31
RandGaussInitAlloc	4-31
RandGaussFree	4-32
RandGaussGetSize	4-33
RandGaussInit	4-34
RandGauss	4-35
RandGauss_Direct	4-36
Special Vector Functions	4-37
VectorJaehne	4-37
VectorRamp	4-38

Chapter 5 **Essential Vector Functions**

Logical and Shift Functions	5-5
AndC	5-6
And	5-7
OrC	5-8
Or	5-9
XorC	5-10
Xor	5-11
Not	5-12
LShiftC	5-13
RShiftC	5-14
Arithmetic Functions	5-15
AddC	5-16
Add	5-17
AddProduct	5-20
AddProductC	5-21
MulC	5-22
Mul	5-24
SubC	5-27
SubCRev	5-28
Sub	5-30

DivC	5-32
DivCRev	5-34
Div	5-35
Abs	5-38
Sqr.....	5-39
Sqrt	5-41
Cubrt	5-44
Exp	5-45
Ln	5-48
10Log10	5-50
SumLn	5-51
Arctan	5-52
Normalize.....	5-53
Conversion Functions	5-54
SortAscend, SortDescend	5-55
SwapBytes	5-56
Convert.....	5-57
Join	5-59
JoinScaled	5-60
SplitScaled	5-61
Conj.....	5-63
ConjFlip	5-64
Magnitude	5-65
MagSquared	5-67
Phase	5-68
PowerSpectr.....	5-69
Real.....	5-70
Imag	5-71
RealToCplx.....	5-72
CplxToReal.....	5-73
Threshold	5-74
Threshold_LT, Threshold_GT	5-77
Threshold_LTAbs, Threshold_GTAbs	5-80
Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal	5-84

Threshold_LTInv	5-88
CartToPolar	5-90
PolarToCart	5-92
MaxOrder	5-94
Preemphasize	5-95
Flip	5-96
FindNearestOne	5-97
FindNearest	5-98
Viterbi Decoder Functions	5-99
GetVarPointDV	5-99
CalcStatesDV	5-100
BuildSymbITableDV4D	5-101
UpdatePathMetricsDV	5-101
Windowing Functions	5-102
Understanding Window Functions	5-102
WinBartlett	5-104
WinBlackman	5-105
WinHamming	5-109
WinHann	5-111
WinKaiser	5-113
Statistical Functions.....	5-115
Sum	5-115
Max	5-117
MaxIndx	5-118
MaxAbs	5-119
MaxAbsIndx	5-120
Min	5-120
MinIndx	5-122
MinAbs	5-123
MinAbsIndx	5-124
MinMax	5-125
MinMaxIndx	5-126
Mean	5-127
StdDev	5-129

Norm	5-131
NormDiff.....	5-133
DotProd	5-136
MaxEvery, MinEvery	5-139
Sampling Functions	5-140
SampleUp	5-140
SampleDown.....	5-142

Chapter 6 **Filtering Functions**

Convolution and Correlation Functions.....	6-1
Conv.....	6-2
ConvBiased.....	6-4
ConvCyclic	6-5
AutoCorr.....	6-5
CrossCorr.....	6-8
UpdateLinear	6-10
UpdatePower	6-11
Filtering Functions	6-13
SumWindow	6-16
FIR Filter Functions.....	6-17
FIRInitAlloc	6-18
FIRMRInitAlloc	6-20
FIRFree	6-23
FIRInit	6-24
FIR MRInit	6-26
FIRGetStateSize, FIRMRGetStateSize	6-29
FIRGetTaps.....	6-32
FIRSetTaps	6-33
FIRGetDlyLine.....	6-35
FIRSetDlyLine.....	6-37
FIROne	6-38
FIR	6-40
FIROne_Direct	6-45
FIR_Direct	6-48

FIRMR_Direct	6-52
FIRSparseInit	6-56
FIRSparseGetStateSize	6-58
FIRSparse	6-59
FIR Filter Coefficient Generating Functions	6-61
FIRGenLowpass	6-61
FIRGenHighpass	6-62
FIRGenBandpass	6-63
FIRGenBandstop	6-65
Single-Rate FIR LMS Filter Functions	6-66
FIRLMSInitAlloc	6-67
FIRLMSFree	6-68
FIRLMSGetTaps	6-68
FIRLMSGetDlyLine	6-69
FIRLMSSetDlyLine	6-70
FIRLMS	6-71
FIRLMSOne_Direct	6-74
Multi-Rate FIR LMS Filter Functions	6-76
FIRLMSMRInitAlloc	6-77
FIRLMSMRFree	6-78
FIRLMSMRSetMu	6-79
FIRLMSMRUpdateTaps	6-80
FIRLMSMRGetTaps	6-81
FIRLMSMRSetTaps	6-82
FIRLMSMRGetTapsPointer	6-83
FIRLMSMRGetDlyLine	6-84
FIRLMSMRSetDlyLine	6-85
FIRLMSMRGetDlyVal	6-86
FIRLMSMRPutVal	6-87
FIRLMSMROne	6-88
FIRLMSMROneVal	6-89
IIR Filter Functions	6-90
IIRInitAlloc	6-92
IIRInitAlloc_BiQuad	6-94

IIRFree	6-96
IIRInit	6-97
IIRInit_BiQuad	6-100
IIRGetStateSize	6-102
IIRGetStateSize_BiQuad	6-103
IIRSetTaps	6-105
IIRGetDlyLine	6-106
IIRSetDlyLine	6-108
IIROne	6-109
IIR	6-111
IIROne_Direct	6-116
IIROne_BiQuadDirect	6-117
IIR_Direct	6-118
IIR_BiQuadDirect	6-119
IIRSparseInit	6-121
IIRSparseGetStateSize	6-122
IIRSparse	6-123
Median Filter Functions	6-126
FilterMedian	6-126

Chapter 7 Transform Functions

Fourier Transform Functions	7-4
Special Arguments	7-4
Flag and Hint Arguments	7-4
Packed Formats	7-5
Pack Format	7-5
Perm Format	7-5
CCS Format	7-6
Format Conversion Functions	7-6
ConjPack	7-7
ConjPerm	7-9
ConjCcs	7-11
Functions for Packed Data Multiplication	7-12
MulPack	7-13

MulPerm.....	7-15
MulPackConj	7-17
Fast Fourier Transform Functions	7-18
FFTInitAlloc_R, FFTInitAlloc_C	7-18
FFTFree_R, FFTFree_C	7-20
FFTInit_R, FFTInit_C	7-22
FFTGetSize_R, FFTGetSize_C	7-24
FFTGetBufSize_R, FFTGetBufSize_C	7-26
FFTFwd_CToC.....	7-27
FFTInv_CToC.....	7-30
FFTFwd_RToPack, FFTFwd_RToPerm, FFTFwd_RToCCS	7-33
FFTInv_PackToR, FFTInv_PermToR, FFTInv_CCSToR.....	7-37
Discrete Fourier Transform Functions.....	7-40
DFTInitAlloc_R, DFTInitAlloc_C	7-40
DFTFree_R, DFTFree_C	7-42
DFTGetBufSize_R, DFTGetBufSize_C	7-43
DFTFwd_CToC	7-45
DFTInv_CToC	7-47
DFTFwd_RToPack, DFTFwd_RToPerm, DFTFwd_RToCCS	7-49
DFTInv_PackToR, DFTInv_PermToR, DFTInv_CCSToR.....	7-52
DFTOutOrdInitAlloc_C	7-54
DFTOutOrdFree_C	7-55
DFTOutOrdGetBufSize_C	7-56
DFTOutOrdFwd_CToC	7-57
DFTOutOrdInv_CToC	7-58
DFT for a Given Frequency (Goertzel) Functions	7-59
Goertz	7-60
GoertzTwo.....	7-62
Discrete Cosine Transform Functions.....	7-63
DCTFwdInitAlloc, DCTInvInitAlloc	7-63
DCTFwdFree, DCTInvFree	7-65
DCTFwdGetBufSize, DCTInvGetBufSize	7-66
DCTFwd, DCTInv	7-67
Hilbert Transform Functions.....	7-70

HilbertInitAlloc	7-70
HilbertFree	7-71
Hilbert	7-72
Wavelet Transform Functions.....	7-74
Transforms for Fixed Filter Banks.....	7-77
WTHaarFwd, WTHaarInv	7-77
Transforms for User Filter Banks.....	7-82
WTFwdInitAlloc, WTInvInitAlloc	7-82
WTFwdFree, WTInvFree	7-85
WTFwd.....	7-86
WTFwdSetDlyLine, WTFwdGetDlyLine	7-89
WTInv.....	7-92
WTInvSetDlyLine, WTInvGetDlyLine	7-95
Wavelet Transforms Example.....	7-98

Chapter 8 **Speech Recognition Functions**

Basic Arithmetics	8-8
AddAllRowSum	8-8
SumColumn	8-9
SumRow.....	8-11
SubRow.....	8-13
CopyColumn_Indirect	8-14
BlockDMatrixInitAlloc	8-16
BlockDMatrixFree	8-17
NthMaxElement	8-18
VecMatMul	8-19
MatVecMul	8-20
Feature Processing.....	8-22
ZeroMean.....	8-22
CompensateOffset	8-23
SignChangeRate	8-25
LinearPrediction	8-26
Durbin	8-28
Schur	8-29

LPToSpectrum	8-31
LPToCepstrum	8-32
CepstrumToLP	8-33
LPToReflection	8-34
ReflectionToLP	8-35
ReflectionToAR	8-36
ReflectionToTilt	8-38
PitchmarkToF0	8-39
UnitCurve	8-40
LPToLSP	8-41
LSPToLP	8-44
MelToLinear	8-45
LinearToMel	8-46
CopyWithPadding	8-47
MelFBankGetSize	8-48
MelFBankInit	8-49
MelFBankInitAlloc	8-50
MelLinFBankInitAlloc	8-53
EmptyFBankInitAlloc	8-56
FBankFree	8-57
FBankGetCenters	8-57
FBankSetCenters	8-58
FBankGetCoeffs	8-59
FBankSetCoeffs	8-60
EvalFBank	8-61
DCTLifterGetSize_MulC0	8-62
DCTLifterInit_MulC0	8-63
DCTLifterInitAlloc	8-64
DCTLifterFree	8-66
DCTLifter	8-66
NormEnergy	8-69
SumMeanVar	8-71
NewVar	8-73
RecSqrt	8-74

AccCovarianceMatrix	8-75
Derivative Functions	8-77
CopyColumn	8-77
EvalDelta.....	8-78
Delta.....	8-80
DeltaDelta	8-85
Pitch Super Resolution.....	8-89
CrossCorrCoeffDecim	8-89
CrossCorrCoeff	8-90
CrossCorrCoeffInterpolation	8-92
Model Evaluation	8-94
AddNRows	8-94
ScaleLM	8-95
LogAdd.....	8-96
LogSub.....	8-97
LogSum	8-99
MahDistSingle.....	8-100
MahDist	8-101
MahDistMultiMix	8-102
LogGaussSingle.....	8-104
LogGauss	8-107
LogGaussMultiMix	8-111
LogGaussMax	8-113
LogGaussMaxMultiMix	8-117
LogGaussAdd	8-120
LogGaussAddMultiMix	8-123
LogGaussMixture	8-125
LogGaussMixtureSelect	8-129
BuildSignTable	8-132
FillShortlist_Row	8-134
FillShortlist_Column	8-136
DTW	8-139
Model Estimation	8-141
MeanColumn	8-142

VarColumn	8-143
MeanVarColumn	8-145
WeightedMeanColumn	8-146
WeightedVarColumn	8-148
WeightedMeanVarColumn	8-149
NormalizeColumn	8-152
NormalizeInRange	8-154
MeanVarAcc.....	8-156
GaussianDist	8-157
GaussianSplit.....	8-159
GaussianMerge	8-160
Entropy	8-161
SinC	8-162
ExpNegSqr	8-163
BhatDist	8-164
UpdateMean	8-166
UpdateVar	8-167
UpdateWeight	8-168
UpdateGConst	8-169
OutProbPreCalc	8-171
DcsClustLAccumulate	8-172
DcsClustLCompute	8-173
Model Adaptation.....	8-175
AddMulColumn	8-175
AddMulRow.....	8-176
QRTransColumn	8-177
DotProdColumn	8-178
MulColumn.....	8-179
SumColumnAbs	8-180
SumColumnSqr	8-181
SumRowAbs	8-182
SumRowSqr.....	8-183
SVD, SVDSort	8-184
WeightedSum	8-186

Vector Quantization.....	8-187
FormVector	8-187
CdbkGetSize	8-190
CdbkInit	8-191
CdbkInitAlloc	8-193
CdbkFree	8-195
GetCdbkSize	8-195
GetCodebook	8-196
VQ	8-197
VQSingle_Sort, VQSingle_Thresh	8-198
SplitVQ	8-200
FormVectorVQ	8-201
Polyphase Resampling	8-204
ResamplePolyphaseInit	8-204
ResamplePolyphaseGetSize	8-205
ResamplePolyphaseSetFilter	8-206
ResamplePolyphaseGetFilter	8-207
ResamplePolyphaseInitAlloc	8-210
ResamplePolyphaseFree	8-212
ResamplePolyphase	8-213
Advanced Aurora Functions.....	8-216
SmoothedPowerSpectrumAurora	8-216
NoiseSpectrumUpdate_Aurora	8-217
WienerFilterDesign_Aurora	8-218
MelFBankInitAlloc_Aurora	8-220
TabsCalculation_Aurora	8-221
ResidualFilter_Aurora	8-222
WaveProcessing_Aurora	8-223
LowHighFilter_Aurora	8-225
HighBandCoding_Aurora	8-226
BlindEqualization_Aurora	8-228
DeltaDelta_Aurora	8-229
VADGetBufSize_Aurora	8-234
VADInit_Aurora	8-235

VADDecision_Aurora	8-235
VADFlush_Aurora	8-236
Ephraim-Malah Noise Suppressor.....	8-238
Noise Suppressor Architecture	8-238
Ephraim-Malah Noise Suppressor Details	8-239
Algorithm Steps.....	8-239
Data Structures	8-242
Filter Update Primitives.....	8-243
FilterUpdateEMNS	8-243
FilterUpdateWiener	8-246
Noise Floor Estimation Primitives	8-248
GetSizeMCRA	8-248
InitMCRA	8-249
AltInitMCRA	8-250
UpdateNoisePSDMCRA	8-251
Acoustic Echo Canceller.....	8-252
Acoustic Echo Canceller Architecture.....	8-252
Frequency Domain Block NLMS Adaptive Filter	8-254
Algorithm Steps.....	8-254
Computational Complexity	8-256
AEC Controller.....	8-257
Algorithm Description.....	8-258
Data Structures.....	8-262
Filter Primitives	8-265
FilterAECNLMS	8-265
Filter Update Primitives.....	8-266
CoefUpdateAECNLMS	8-266
Step Size Update Primitives	8-268
StepSizeUpdateAECNLMS	8-268
AEC Controller Primitives	8-269
ControllerGetSizeAEC	8-269
ControllerInitAEC	8-270
ControllerUpdateAEC	8-271
Voice Activity Detector.....	8-272

Voice Activity Detector Architecture	8-273
Voice Activity Detection Primitives	8-274
FindPeaks	8-274
PeriodicityLSPE	8-275
Periodicity	8-276

Chapter 9 Speech Coding Functions

Rounding mode	9-2
Notational Conventions	9-3
Definitions	9-3
Data Structures	9-4
Common Functions	9-5
ConvPartial	9-6
InterpolateC_NR	9-7
Mul_NR	9-8
MulC_NR	9-9
MulPowerC_NR	9-11
AutoScale	9-12
DotProdAutoScale	9-13
InvSqrt	9-14
AutoCorr	9-14
AutoCorrLagMax	9-15
AutoCorr_NormE	9-17
CrossCorr	9-18
CrossCorrLagMax	9-19
SynthesisFilter	9-20
G.729 Related Functions	9-22
Basic Functions	9-24
DotProd_G729	9-24
Interpolate_G729	9-25
Linear Prediction Analysis Functions	9-27
AutoCorr_G729	9-27
LevinsonDurbin_G729	9-28
LPCToLSP_G729	9-30

LSFToLSP_G729	9-32
LSFQuant_G729	9-32
LSFDecode_G729	9-34
LSFDecodeErased_G729	9-35
LSPToLPC_G729	9-36
LSPQuant_G729	9-37
LSPToLSF_G729	9-40
LagWindow_G729	9-41
Codebook Search Functions	9-43
OpenLoopPitchSearch_G729	9-43
AdaptiveCodebookSearch_G729	9-46
DecodeAdaptiveVector_G729	9-49
FixedCodebookSearch_G729	9-50
GainCodebookSearch	9-54
ToeplizMatrix_G729	9-55
Codebook Gain Functions	9-58
DecodeGain_G729	9-58
GainControl_G729	9-59
GainQuant_G729	9-61
AdaptiveCodebookContribution_G729	9-63
AdaptiveCodebookGain_G729	9-64
Filter Functions	9-66
ResidualFilter_G729	9-67
SynthesisFilter_G729	9-68
LongTermPostFilter_G729	9-70
ShortTermPostFilter_G729	9-73
TiltCompensation_G729	9-75
HarmonicFilter	9-77
HighPassFilterSize_G729	9-78
HighPassFilterInit_G729	9-79
HighPassFilter_G729	9-80
IIR16s_G729	9-81
PhaseDispersionGetStateSize_G729D	9-82
PhaseDispersionInit_G729D	9-82

PhaseDispersionUpdate_G729D	9-83
PhaseDispersion_G729D	9-84
Preemphasize_G729A	9-85
WinHybridGetStateSize_G729E	9-86
WinHybridInit_G729E	9-86
WinHybrid_G729E	9-87
RandomNoiseExcitation_G729B	9-88
FilteredExcitation_G729	9-89
G.723.1 Related Functions	9-90
Linear Prediction Analysis Functions	9-91
AutoCorr_G723	9-92
AutoCorr_NormE_G723	9-93
LevinsonDurbin_G723	9-94
LPCToLSF_G723	9-96
LSFToLPC_G723	9-97
LSFDecode_G723	9-98
LSFQuant_G723	9-99
Codebook Search Functions.....	9-100
OpenLoopPitchSearch_G723	9-100
ACELPFixedCodebookSearch_G723	9-101
AdaptiveCodebookSearch_G723	9-103
MPMLQFixedCodebookSearch_G723	9-104
ToeplitzMatrix_G723	9-106
Gain Quantization	9-107
GainQuant_G723	9-107
GainControl_G723	9-109
Filter Functions.....	9-110
HighPassFilter_G723	9-110
IIR16s_G723	9-111
SynthesisFilter_G723	9-112
TiltCompensation_G723	9-114
HarmonicSearch_G723	9-114
HarmonicNoiseSubtract_G723	9-116
DecodeAdaptiveVector_G723	9-117

PitchPostFilter_G723	9-118
GSM-AMR Related Functions	9-121
Basic Functions	9-123
Interpolate_GSMAMR	9-123
FFTFwd_RToPerm_GSMAMR	9-124
LP Analysis and Quantization Primitives	9-125
AutoCorr_GSMAMR	9-125
LevinsonDurbin_GSMAMR	9-127
LPCToLSP_GSMAMR	9-129
LSPToLPC_GSMAMR	9-130
LSFToLSP_GSMAMR	9-132
LSPQuant_GSMAMR	9-132
QuantLSPDecode_GSMAMR	9-135
Adaptive Codebook Primitives	9-136
Open-Loop Pitch Search (OLP).....	9-137
OpenLoopPitchSearchNonDTX_GSMAMR	9-139
OpenLoopPitchSearchDTXVAD1_GSMAMR	9-141
OpenLoopPitchSearchDTXVAD2_GSMAMR	9-144
ImpulseResponseTarget_GSMAMR	9-147
AdaptiveCodebookSearch_GSMAMR	9-149
AdaptiveCodebookDecode_GSMAMR	9-152
AdaptiveCodebookGain_GSMAMR	9-154
Fixed Codebook Search	9-155
AlgebraicCodebookSearch_GSMAMR	9-156
FixedCodebookDecode_GSMAMR	9-159
Discontinuous Transmission (DTX).....	9-160
Preemphasize_GSMAMR	9-160
VAD1_GSMAMR	9-161
VAD2_GSMAMR	9-164
EncDTXSID_GSMAMR	9-166
EncDTXHandler_GSMAMR	9-168
EncDTXBuffer_GSMAMR, DecDTXBuffer_GSMAMR	9-170
Post Processing	9-171
PostFilter_GSMAMR	9-171

AMR Wideband Related Functions.....	9-174
ResidualFilter_AMRWB	9-176
LPC Analysis Functions	9-177
LPCToISP_AMRWB	9-177
ISPToLPC_AMRWB	9-178
ISPToISF_Norm_AMRWB	9-180
ISFToISP_AMRWB	9-181
Open-loop Pitch Analysis Functions	9-182
OpenLoopPitchSearch_AMRWB	9-182
Filtering Functions.....	9-184
HighPassFilterGetSize_AMRWB	9-184
HighPassFilterInit_AMRWB	9-185
HighPassFilter_AMRWB	9-186
Preemphasize_AMRWB	9-187
Deemphasize_AMRWB	9-188
SynthesisFilter_AMRWB	9-189
Discontinuous Transmission (DTX) Functions	9-190
VADGetSize_AMRWB	9-190
VADInit_AMRWB	9-191
VAD_AMRWB	9-191
Codebook Search Functions.....	9-192
AlgebraicCodebookSearch_AMRWB	9-192
AlgebraicCodebookDecode_AMRWB	9-197
AdaptiveCodebookGainCoeff_AMRWB	9-198
AdaptiveCodebookSearch_AMRWB	9-200
AdaptiveCodebookDecodeGetSize_AMRWB	9-203
AdaptiveCodebookDecodeInit_AMRWB	9-203
AdaptiveCodebookDecodeUpdate_AMRWB	9-204
AdaptiveCodebookDecode_AMRWB	9-205
Quantization Functions	9-206
ISFQuant_AMRWB	9-206
ISFQuantDecode_AMRWB	9-208
ISFQuantDTX_AMRWB	9-209
ISFQuantDecodeDTX_AMRWB	9-210

GainQuant_AMRWB	9-211
DecodeGain_AMRWB	9-213
EncDTXBuffer_AMRWB	9-215
DecDTXBuffer_AMRWB	9-216
GSM Full Rate Related Functions	9-218
RPEQuantDecode_GSMFR	9-218
Deemphasize_GSMFR	9-219
ShortTermAnalysisFilter_GSMFR	9-220
ShortTermSynthesisFilter_GSMFR	9-221
HighPassFilter_GSMFR	9-222
Schur_GSMFR	9-223
WeightingFilter_GSMFR	9-224
Preemphasize_GSMFR	9-225
G.722.1 Related Functions	9-226
DCTFwd_G722, DCTInv_G722	9-226
DecomposeMLTToDCT	9-227
DecomposeDCTToMLT	9-229
HuffmanEncode_G722	9-230
G.726 Related Functions	9-231
EncodeGetStateSize_G726	9-231
EncodeInit_G726	9-232
Encode_G726	9-233
DecodeGetStateSize_G726	9-234
DecodeInit_G726	9-234
Decode_G726	9-235
G.728 Related Functions	9-237
IIRGetStateSize_G728	9-238
IIR_Init_G728	9-238
IIR_G728	9-239
SynthesisFilterGetStateSize_G728	9-240
SynthesisFilterInit_G728	9-240
SyntesisFilter_G728	9-241
CombinedFilterGetStateSize_G728	9-242
CombinedFilterInit_G728	9-243

CombinedFilter_G728	9-243
PostFilterGetStateSize_G728	9-245
PostFilterInit_G728	9-245
PostFilter_G728	9-246
PostFilterAdapterGetStateSize_G728	9-247
PostFilterAdapterStateInit_G728	9-248
LPCInverseFilter_G728	9-249
PitchPeriodExtraction_G728	9-250
WinHybridGetStateSize_G728	9-252
WinHybridInit_G728	9-253
WinHybrid_G728	9-254
LevinsonDurbin_G728	9-255
CodebookSearch_G728	9-257
ImpulseResponseEnergy_G728	9-258
Echo Canceller Related Functions	9-259
SubbandProcessGetSize	9-260
SubbandProcessInit	9-261
SubbandAnalysis	9-262
SubbandSynthesis	9-263
SubbandControllerGetSize_EC	9-265
SubbandControllerInit_EC	9-266
SubbandControllerUpdate_EC	9-267
SubbandController_EC	9-268
SubbandControllerReset_EC	9-269
ToneDetectGetStateSize_EC	9-270
ToneDetectInit_EC	9-271
ToneDetect_EC	9-272
FullbandControllerGetSize_EC	9-273
FullbandControllerInit_EC	9-274
FullbandControllerUpdate_EC	9-275
FullbandController_EC	9-276
FullbandControllerReset_EC	9-277
FIR_EC	9-278
NLMS_EC	9-279

G722 Sub-Band ADPCM Speech Codec Related Functions	9-280
SBADPCMEncodeStateSize_G722	9-280
SBADPCMEncodeInit_G722	9-281
SBADPCMEncode_G722	9-282
QMFEncode_G722	9-283
SBADPCMDecodeStateSize	9-284
SBADPCMDecodeInit_G722	9-284
SBADPCMDecode_G722	9-285
QMFDecode_G722	9-286
Companding Functions	9-288
MuLawToLin	9-288
LinToMuLaw	9-289
ALawToLin	9-290
LinToALaw	9-291
MuLawToALaw	9-293
ALawToMuLaw	9-294

Chapter 10 Audio Coding Functions

Interleaved to Multi-Row Format Conversion Functions	10-7
Interleave	10-7
Deinterleave	10-8
Spectral Data Prequantization Functions	10-10
Pow34	10-10
Pow43	10-12
Scale Factors Calculation Functions	10-13
CalcSF	10-13
Mantissa Conversion and Scaling Functions	10-14
Scale	10-15
MakeFloat	10-16
Modified Discrete Cosine Transform Functions	10-17
MDCTFwdInitAlloc, MDCTInvInitAlloc	10-17
MDCTFwdInit, MDCTInvInit	10-18
MDCTFwdFree, MDCTInvFree	10-20
MDCTFwdGetSize, MDCTInvGetSize	10-21

MDCTFwdGetBufSize, MDCTInvGetBufSize	10-22
MDCTFwd, MDCTInv	10-23
Block Filtering Functions.....	10-25
FIRBlockInitAlloc	10-25
FIRBlockFree	10-26
FIRBlockOne	10-27
Frequency Domain Prediction Functions	10-29
FDPInitAlloc	10-29
FDPInit	10-30
FDPFree	10-31
FDPGetSize	10-31
FDPReset	10-32
FDPResetSfb	10-33
FDPResetGroup	10-34
FDPFwd	10-34
FDPInv	10-35
VLC Functions	10-37
VLCDecodeEscBlock_MP3	10-37
VLCDecodeEscBlock_AAC	10-40
VLCCountEscBits_MP3	10-42
VLCCountEscBits_AAC	10-43
VLCEncodeEscBlock_MP3	10-44
VLCEncodeEscBlock_AAC	10-46
Psychoacoustic Functions	10-47
Spread	10-47
Vector Quantization Functions	10-48
VQCodeBookInitAlloc	10-48
VQCodeBookInit	10-49
VQCodeBookFree	10-50
VQCodeBookGetSize	10-50
VQPreliminarySelect	10-51
VQMainSelect	10-53
VQIndexSelect	10-55
VQReconstruction	10-57

MP3 Audio Coding Functions	10-59
Macros and Constants	10-59
Data Structures	10-60
Frame Header	10-60
Side Information	10-61
MP3 Huffman Table Structure	10-62
MP3 Psychoacoustic Model Two Analysis	10-62
Psychoacoustic Model Two State.....	10-62
MP3 Bit Reservoir	10-64
MP3 Codec Enumerated Types	10-65
MP3 Audio Encoder	10-66
AnalysisPQMF_MP3	10-67
MDCTFwd_MP3	10-68
PsychoacousticModelTwo_MP3	10-70
JointStereoEncode_MP3	10-75
Quantize_MP3	10-78
PackScalefactors_MP3	10-84
HuffmanEncode_MP3	10-86
PackFrameHeader_MP3	10-89
PackSideInfo_MP3	10-90
BitReservoirInit_MP3	10-92
MP3 Audio Decoder	10-94
UnpackFrameHeader_MP3	10-95
UnpackSideInfo_MP3	10-96
UnpackScaleFactors_MP3	10-97
HuffmanDecode_MP3	
HuffmanDecodeSfb_MP3	
HuffmanDecodeSfbMbp_MP3.....	10-99
ReQuantize_MP3	
ReQuantizeSfb_MP3.....	10-101
MDCTInv_MP3	10-103
SynthPQMF_MP3	10-105
Advanced Audio Coding Functions.....	10-108
Global Macros.....	10-108

Data Types and Structures	10-109
ADIF Header	10-109
ADTS Frame Header	10-110
Individual Channel Side Information.....	10-111
AAC Scalable Main Element Header.....	10-111
AAC Scalable Extension Element Header.....	10-112
AAC Program Config Element.....	10-112
Temporal Noise Shaping (TNS) Structure for One Layer	10-114
Long Term Prediction (LTP) Structure	10-115
Channel Pair Element.....	10-115
Channel Information	10-116
AAC Decoding Primitives	10-118
MPEG-2 AAC Primitives.....	10-119
UnpackADIFHeader_AAC	10-119
UnpackADTSFrameHeader_AAC	10-120
DecodePrgCfgElt_AAC	10-121
DecodeChanPairElt_AAC	10-122
NoiselessDecoder_LC_AAC	10-123
DecodeDatStrElt_AAC	10-127
DecodeFillElt_AAC	10-128
QuantInv_AAC	10-129
DecodeMsStereo_AAC	10-131
DecodelsStereo_AAC	10-133
DeinterleaveSpectrum_AAC	10-135
DecodeTNS_AAC	10-136
MDCTInv_AAC	10-140
MPEG-4 AAC Primitives.....	10-142
DecodeMainHeader_AAC	10-142
DecodeExtensionHeader_AAC	10-143
DecodePNS_AAC	10-145
DecodeMsPNS_AAC	10-146
DecodeChanPairElt_MP4_AAC	10-147
LongTermReconstruct_AAC	10-149
MDCTFwd_AAC	10-150

EncodeTNS_AAC	10-151
LongTermPredict_AAC	10-152
NoiselessDecode_AAC	10-153
LtpUpdate_AAC	10-155
Spectral Band Replication Functions.....	10-157
QMF Functions	10-158
Size Getting	10-159
AnalysisFilterGetSize_SBR	10-159
SynthesisFilterGetSize_SBR	10-160
SynthesisDownFilterGetSize_SBR	10-161
Intialization	10-162
AnalysisFilterInit_SBR	10-162
SynthesisFilterInit_SBR	10-163
SynthesisDownFilterInit_SBR	10-165
QMF Main Functions	10-166
AnalysisFilter_SBR	10-166
SynthesisFilter_SBR	10-169
SynthesisDownFilter_SBR	10-172
Prediction	10-175
PredictCoef_SBR	10-175

Chapter 11 String Functions

String Manipulation.....	11-2
Find, FindRev	11-3
FindC, FindRevC	11-5
FindCAny, FindRevCAny	11-7
Insert.....	11-8
Remove.....	11-9
Compare	11-11
CompareIgnoreCase, CompareIgnoreCaseLatin	11-13
Equal	11-14
TrimC	11-15
TrimCAny, TrimStartCAny, TrimEndCAny	11-16
ReplaceC	11-17

Uppercase, UppercaseLatin	11-18
Lowercase, LowercaseLatin	11-19
Hash	11-21
Concat	11-22
ConcatC	11-23
SplitC	11-24
Regular Expressions.....	11-25
RegExInitAlloc	11-26
RegExpFree	11-27
RegExInit	11-28
RegExpGetSize	11-29
RegExpSetMatchLimit	11-30
RegExpFind	11-31

Chapter 12 Fixed-Accuracy Arithmetic Functions

Power and Root Functions.....	12-4
Inv	12-4
Div	12-6
Sqrt	12-8
InvSqrt	12-10
Cbrt	12-12
InvCbrt	12-13
Pow	12-15
Powx	12-18
Exponential and Logarithmic Functions.....	12-21
Exp	12-21
Ln	12-23
Log10	12-25
Trigonometric Functions.....	12-27
Cos	12-27
Sin	12-29
SinCos	12-30
Tan	12-32
Acos	12-34

Asin	12-36
Atan	12-38
Atan2	12-40
Hyperbolic Functions	12-43
Cosh	12-43
Sinh	12-45
Tanh	12-47
Acosh	12-48
Asinh	12-50
Atanh	12-52
Special Functions	12-54
Erf	12-54
Erfc	12-56

Chapter 13 Data Compression Functions

Application Notes.....	13-1
VLC and Huffman Coding Functions	13-3
Data Compression VLC functions.....	13-4
VLCEncodeInitAlloc	13-5
VLCEncodeFree	13-6
VLCEncodeInit	13-6
VLCEncodeGetSize	13-7
VLCEncodeBlock	13-8
VLCEncodeOne	13-9
VLCCountBits	13-10
VLCDecodeInitAlloc	13-11
VLCDecodeFree	13-12
VLCDecodeInit	13-12
VLCDecodeGetSize	13-14
VLCDecodeBlock	13-15
VLCDecodeOne	13-16
Example of Using VLC Coding Functions	13-17
Huffman Coding	13-18

EncodeHuffInitAlloc	13-19
HuffFree	13-20
EncodeHuffInit	13-20
HuffGetSize	13-21
EncodeHuffOne	13-22
EncodeHuff	13-23
EncodeHuffFinal	13-24
HuffGetLenCodeTable	13-24
DecodeHuffInitAlloc	13-25
DecodeHuffInit	13-26
DecodeHuffOne	13-27
DecodeHuff	13-27
HuffGetDstBuffSize	13-28
Application Notes	13-29
HuffLenCodeTablePack	13-29
HuffLenCodeTableUnpack	13-30
Dictionary-Based Compression Functions.....	13-32
LZSS Compression Functions	13-33
EncodeLZSSInitAlloc	13-34
LZSSFree	13-34
EncodeLZSSInit	13-35
LZSSGetSize	13-36
EncodeLZSS	13-36
EncodeLZSSFlush	13-37
DecodeLZSSInitAlloc	13-40
DecodeLZSSInit	13-41
DecodeLZSS	13-42
ZLIB Coding Functions.....	13-45
Special Parameters	13-48
EncodeLZ77Init	13-50
EncodeLZ77GetSize	13-51
EncodeLZ77InitAlloc	13-52
LZ77Free	13-53
EncodeLZ77	13-53

EncodeLZ77SelectHuffMode	13-54
EncodeLZ77FixedHuff	13-55
EncodeLZ77DynamicHuff	13-57
EncodeLZ77StoredBlock	13-58
EncodeLZ77Flush	13-59
EncodeLZ77GetPairs	13-60
EncodeLZ77SetPairs	13-61
EncodeLZ77GetStatus	13-61
EncodeLZ77SetStatus	13-62
EncodeLZ77Reset	13-63
DecodeLZ77Init	13-64
DecodeLZ77GetSize	13-64
DecodeLZ77InitAlloc	13-65
DecodeLZ77	13-66
DecodeLZ77GetBlockType	13-67
DecodeLZ77FixedHuff	13-68
DecodeLZ77DynamicHuff	13-69
DecodeLZ77StoredBlock	13-71
DecodeLZ77GetPairs	13-72
DecodeLZ77SetPairs	13-73
DecodeLZ77GetStatus	13-73
DecodeLZ77SetStatus	13-74
DecodeLZ77Reset	13-75
Adler32	13-75
CRC32	13-76
Example of Using Intel IPP ZLIB Coding Functions	13-77
Implementing the deflate and inflate Functions	13-79
BWT-Based Compression Functions	13-92
Burrows-Wheeler Transform	13-93
BWTFwdGetSize	13-95
BWTFwd	13-95
BWTInvGetSize	13-96
BWTInv	13-97
BWTGetSize_SmalBlock	13-98

BWTFwd_SmallBlock	13-98
BWTInv_SmallBlock	13-99
Generalized Interval Transformation Coding	13-100
Special Arguments	13-101
EncodeGITInitAlloc	13-101
GITFree	13-102
EncodeGITInit	13-103
EncodeGITGetSize	13-104
EncodeGIT	13-105
DecodeGITInitAlloc	13-109
DecodeGITInit	13-110
DecodeGITGetSize	13-111
DecodeGIT	13-111
Move To Front Functions	13-115
MTFInitAlloc	13-116
MTFFree	13-117
MTFInit	13-117
MTFGetSize	13-118
MTFFwd	13-118
MTFInv	13-119
Run Length Encoding Functions	13-120
EncoderRLE	13-120
DecoderRLE	13-121

Appendix A Handling of Special Cases

Appendix B Removed Functions

Bibliography

Glossary

Index

Overview

1

This document describes the structure, operation and functions of the Intel® Integrated Performance Primitives (Intel® IPP) for Intel® architecture that operate on one-dimensional signals. This is the first volume of the Intel IPP Reference Manual, which also comprises descriptions of the Intel IPP for image and video processing (volume 2), operations on small matrices (volume 3), and cryptography functions (volume 4).

The Intel IPP software package supports many functions whose performance can be significantly enhanced on Intel architecture, particularly using the MMX™ technology and Streaming SIMD Extensions (SSE). For information on Intel IPP implementation for Intel® PCA application processors, see [Cross-Architecture Alignment](#) section later in this chapter.

The Intel IPP for signal processing software is a collection of low-overhead, high- performance operations performed on one-dimensional (1D) data arrays.

This manual explains the Intel IPP concepts as well as specific data type definitions and operation models used in the signal processing domain and provides detailed descriptions of the Intel IPP signal processing functions.

This chapter introduces the Intel IPP software and explains the organization of this manual.

About This Software

The Intel IPP for Intel architecture software enables taking advantage of the parallelism of the single-instruction, multiple-data (SIMD) instructions that comprise the core of the MMX technology and Streaming SIMD Extensions. These technologies improve the performance of computation-intensive signal, image, and video processing applications.

Hardware and Software Requirements

The Intel IPP for Intel architecture runs on personal computers that are based on IA-32 processors or Itanium® architecture-based processors and running Microsoft Windows* 2000, Windows ME, Windows XP, or Linux*. The Intel IPP integrates into the customer's application or library written in C or C++.

Platforms Supported

The Intel IPP for Intel architecture software runs on Windows and Linux platforms. The code and syntax used for function and variable declarations in this manual are written in the ANSI C style. However, versions of the Intel IPP for different processors or operating systems may, of necessity, vary slightly.

Cross-Architecture Alignment

Cross Architecture Overview

Intel IPP has been designed to support application development on various Intel® architectures. Previously Intel IPP was offered in two separate products, one for the Intel® Pentium®, Xeon® and Itanium® processors and one for the Intel® PCA processors based on Intel XScale® technology. While these separate packages included many similarities, there were some differences in both functionality and interface.

With rising interest in cross architecture development, the Intel IPP development teams have worked to bridge these differences to allow for easy development of applications for multiple Intel® platforms. Starting from version 4.0, Intel IPP represents the result of this effort and provides alignment of the two separate packages into a single offering that includes support for all of these architectures. This includes interface alignment and full API alignment for all functions that are relevant to both architectures.

Currently, the functions available for Intel Pentium, Xeon and Itanium processors represent a superset of the functions available for the Intel PCA processors. This means the API definition is common for all processors, while the underlying function implementation takes into account the variations in processor architectures.

By providing a single cross-architecture API, Intel IPP allows software application repurposing and enables developers to port to unique features across Intel® processor-based desktop, server, mobile and handheld platforms. Developers can write their code once, to realize application performance over many processor generations.

For additional information on API additions and changes from previous versions, please refer to the product release notes and the document *Migration Guide for Intel® IPP Cross Architecture API Alignment* available at <http://www.intel.com/software/products/ipp>.

The following table summarizes the functionality covered in each Intel IPP implementation.

Table 1-1 Function Coverage in Intel IPP

Function Group	Intel® Pentium® 4 processors	Intel® Xeon® processors with Intel® EM64T	Intel® Itanium® 2 processors	Intel® PCA processors	Intel® IXP4XX Product Line
Signal Processing	available	available	available	available ¹	available
Image Processing	available	available	available	available ¹	available
JPEG	available	available	available	available ¹	available
Speech Recognition	available	available	available	n/a	available
Speech Coding	available	available	available	available ¹	available
Audio Codecs	available	available	available	available ¹	available
Video Codecs	available	available	available	available ¹	available
Matrix	available	available	available	n/a	n/a
Vector Math	available	available	available	n/a	n/a
Computer Vision	available	available	available	n/a	available
Cryptography	available	available	available	available	available
Data Compression	available	available	available	n/a	available
Color Conversion	available	available	available	n/a	available
String Processing	available	available	available	n/a	available

1. Intel PCA processors support a subset of these functions.

API Changes in Version 5.0

Starting from the version 5.0 Intel IPP has a limited compatibility with the versions 4.x or lower. To improve the library in several aspects the following changes have been done:

- some functions have been replaced by new functions with extended functionality;
- some functions have been changed to make API more consistent and handy;
- odd and unusable functions have been discarded;

- several complicated functions have been discarded from the Intel IPP, but their functionalities have been moved to the codec/sample level.

All these changes affect existent applications. [Table 1](#) in the [Appendix B](#) lists such functions for the signal processing functional domains and specifies the corresponding Intel IPP 5.0 functions to substitute for them. If an application calls functions listed in this table, then the source code should be modified.

Several functions have been moved to the newly created domains. These changes do not affect existent applications.

Additionally, certain modifications breaking binary compatibility have been made in the Intel IPP 5.0, for example, directory tree structure has been modified, the `ipp20` folder has been discarded. These changes also affect existent applications that should be at least rebuilt.

The version 5.1 as well as the next following versions has full backward compatibility with the version 5.0.

Technical Support

Intel IPP provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: <http://developer.intel.com/software/products/>.

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, and more (visit <http://support.intel.com/support/>).

Registering your product entitles you to one-year technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing the following services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, or contact Intel, or seek product support, please visit: <http://www.intel.com/software/products/support>

Intel IPP Code Samples

An extensive library of code samples and codecs have been implemented using the Intel IPP functions to help demonstrate the use of Intel IPP and to help accelerate the development of your application, components and codecs. The samples can be downloaded from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

About This Manual

This manual provides a background for the signal processing concepts used in the Intel IPP software as well as detailed descriptions of the Intel IPP for Intel architecture signal processing functions.



NOTE. In the following discussion throughout the manual, Intel IPP always refers to the Intel Integrated Performance Primitives for Intel architecture, unless specifically stated otherwise. For description of Intel PCA version of Intel IPP, refer to the Intel IPP for Intel XScale® microarchitecture Reference Manual.

The Intel IPP functions are combined in groups by their functionality. Each group of functions is described in a separate chapter (chapters 3 through 12).

Manual Organization

This manual contains the following chapters:

- Chapter 1 “[Overview](#).” Introduces the Intel IPP software, provides information on manual organization, and explains notational conventions.
- Chapter 2 “[Intel Integrated Performance Primitives Concepts](#).” Explains the basic concepts underlying signal processing in Intel IPP and describes the supported data formats and operation modes.
- Chapter 3 “[Support Functions](#).” Describes functions used to manipulate the signals, such as Copy and Set. Also describes data conversion and memory allocation functions.
- Chapter 4 “[Vector Initialization Functions](#).” Describes initialization and sample-generating functions.
- Chapter 5 “[Essential Vector Functions](#).” Describes vector manipulation functions.
- Chapter 6 “[Filtering Functions](#).” Details filtering operations that use linear and non-linear filters.

- Chapter 7 [“Transform Functions.”](#) Describes domain transform functions (Fourier, Wavelet, Cosine).
- Chapter 8 [“Speech Recognition Functions.”](#) Describes functions used in the speech recognition applications.
- Chapter 9 [“Speech Coding Functions.”](#) Describes functions used as buiding blocks for implementing speech codecs.
- Chapter 10 [“Audio Coding Functions.”](#) Includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio codec, MP3 encoder and decoder, as well as for implementation of a portable optimized MPEG-4 AAC Main profile decoder and a portable optimized MPEG-1, 2 Layer III encoder.
- Chapter 11 [“String Functions.”](#) Describes functions that operate on strings of symbols.
- Chapter 12 [“Fixed-Accuracy Arithmetic Functions.”](#) Describes Intel IPP fixed-accuracy vector mathematical functions suitable for multimedia and signal processing in real time applications.
- Chapter 13 [“Data Compression Functions.”](#) Describes Intel IPP data compression functions that support a number of different conversion methods.

The manual also includes an [Appendix](#), a [Glossary](#) of terms, a [Bibliography](#), and an [Index](#).

Function Descriptions

In Chapters 3 through 12, each function is introduced by its short name (without the `ipps` prefix and modifiers) and a brief description of its purpose. This is followed by the function call sequence, definition of its parameters (arguments), and more detailed explanation of the function’s purpose. The following sections are included in function description:

<i>Syntax</i>	Lists function prototypes
<i>Parameters</i>	Describes all the function parameters (arguments).
<i>Description</i>	Defines the function and describes the operation performed by the function. This section may also include the code examples and descriptive equations.
<i>Application Notes</i>	If present, describes any special information which application programmers or other users of the function need to know.
<i>Return Values</i>	Explains the value returned by the function. Most commonly, it lists error codes that the function returns.
<i>See Also</i>	If present, lists the names of functions which perform related tasks.

Audience for This Manual

The manual is intended for the developers of signal processing applications and libraries, as well as cross-domain applications. The audience is expected to be experienced in using C and to have a working knowledge of the vocabulary and principles of signal processing.

Online Version

This manual is available in an electronic format (Portable Document Format, or PDF). To obtain a hard copy of the manual, print the file using the printing capability of Adobe Acrobat*, the tool used for the online presentation of the document.

Related Publications

For more information about signal processing concepts and algorithms, refer to the books and papers listed in the [Bibliography](#).

Notational Conventions

In this manual, notational conventions include:

- Fonts used for distinction between the text and the code;
- Signal name conventions;
- Naming conventions for different items.

Font Conventions

The following font conventions are used throughout the manual:

<code>THIS TYPE STYLE</code>	Used in the text for Intel IPP constant identifiers; for example, <code>IPP_MAX_64S</code> .
<code>This type style</code>	Mixed with the uppercase in structure names as in <code>IppLibraryVersion</code> ; also used in function names, code examples and call statements; for example, <code>void ippsFree()</code> .
<i>This type style</i>	Variables in arguments and parameters description; for example, <i>val</i> , <i>srcLen</i> .

Signal Name Conventions

In this manual, vectors and arrays are commonly used to represent a discrete 1D signal. The notation $x(n)$ refers to a conceptual signal, while the notation $x[n]$ refers to an actual vector. Both of these are annotated to indicate a specific finite range of values:

$$x[n], \quad 0 \leq n < len.$$

Typically, the number of elements in vectors is denoted by *len*. Vector names contain square brackets as distinct from vector elements with current index *n*.

For example, the expression $pDst[n] = pSrc[n] + val$ implies that each element $pDst[n]$ of the vector *pDst* is computed for each *n* in the range from 0 to *len*-1. Special cases are regarded and described separately.

Naming Conventions

The following naming conventions for different items are used by the Intel IPP software:

- Constant identifiers are in uppercase; for example, `IPP_MIN_64S`.
- All structures and enumerators, specific for the signal processing domain have the `Ipps` prefix, while those common for entire Intel IPP software have the `IPP` prefix; for example, `IppsROI`, `IppLibraryVersion`.
- All names of the functions used for signal processing have the `ipps` prefix. In code examples, you can distinguish the Intel IPP interface functions from the application functions by this prefix.



NOTE. In this manual, the `ipps` prefix in function names is always used in the code examples. In the text, this prefix is usually omitted when referring to the function group.

- Each new part of a function name starts with an uppercase character, without underscore; for example, `ippsAddAllRowSum`.

For the detailed description of function name structure in Intel IPP, see “Function Naming” on [page 2-2](#).

Intel® Integrated Performance Primitives Concepts

2

This chapter explains the purpose and structure of the Intel® Integrated Performance Primitives (Intel® IPP) software and looks over some of the basic concepts used in the signal processing part of Intel IPP. It also describes the supported data formats and operation modes, and defines function naming conventions in the manual.

Basic Features

The Intel Integrated Performance Primitives, like other members of the Intel® Performance Libraries, is a collection of high-performance code that performs domain-specific operations. It is distinguished by providing a low-level, stateless interface.

Based on experience in developing and using Intel Performance Libraries, Intel IPP has the following major distinctive features:

- The Intel IPP provides basic low-level functions for creating applications in several different domains, such as signal processing, image and video processing, operations on small matrices, and cryptography applications;
- Intel IPP functions follow the same interface conventions including uniform naming rules and similar composition of prototypes for primitives that refer to different application domains;
- Intel IPP functions use abstraction level which is best suited to achieve superior performance figures by the application programs.

To speed up performance, Intel IPP functions are optimized to use all benefits of Intel® architecture processors. Besides that, most of Intel IPP functions do not use complicated data structures, which helps reduce overall execution overhead.

Intel IPP is well-suited for cross-platform applications. For example, the functions developed for IA-32 platform can be readily ported to Itanium[®]-based platforms and systems with Intel[®] StrongARM* technology or Intel XScale[®] technology. For more information on platform compatibility, see [Cross-Architecture Alignment](#) section in chapter 1.

Function Naming

Naming conventions for the Intel IPP functions are similar for all covered domains. You can distinguish signal processing functions by the `ipps` prefix, while image and video processing functions have `ippi` prefix, and functions that are specific for operations on small matrices have `ippm` prefix in their names.

Function names in Intel IPP have the following general format:

```
ipp<data-domain><name>_<datatype>[_<descriptor>]( <arguments> );
```

The elements of this format are explained in the sections that follow.

Data-Domain

The *data-domain* is a single character that expresses the subset of functionality to which a given function belongs. The current version of Intel IPP supports the following data-domains:

- s for signals (expected data type is a 1D signal);
- i for images and video (expected data type is a 2D image);
- m for matrices (expected data type is a matrix).

For example, function names that begin with `ipps` signify that respective functions are used for signal processing.

Name

The *name* is an abbreviation for the core operation that the function really does, for example `Set`, `Copy`, followed in some cases by a function-specific modifier:

```
<name> = <operation>[_modifier]
```

This modifier, if present, denotes a slight modification or variation of the given function. For example, the modifier `CToC` in the function `ippsFFTInv_CToC_32fc` signifies that the inverse fast Fourier transform operates on complex data, performing Complex-To-Complex (CToC) transform.

Data Types

The *datatype* field indicates data types used by the function, in the following format:

<bit depth><bit interpretation> ,

where

bit depth = <1|8|16|32|64>

and

bit interpretation = <*u*|*s*|*f*>[*c*]

Here *u* indicates “unsigned integer”, *s* indicates “signed integer”, *f* indicates “floating point”, and *c* indicates “complex”.

The current version of Intel IPP supports the data types of the source and destination for signal processing functions listed in [Table 2-1](#).



NOTE. In the lists of function arguments, the `Ipp` prefix is written in the data type. For example, the 8-bit signed data is denoted as `Ipp8s` type. These Intel IPP -specific data types are defined in the respective library header files.

Table 2-1 Data Types Supported by Intel IPP for Signal Processing

Type	Usual C Type	Intel IPP Type
8u	unsigned char	Ipp8u
8s	signed char	Ipp8s
16u	unsigned short	Ipp16u
16s	signed short	Ipp16s
16sc	complex short	Ipp16sc
32u	unsigned int	Ipp32u
32s	signed int	Ipp32s
32f	float	Ipp32f
32fc	complex float	Ipp32fc
64s	__int64 (Windows) or long long (Linux)	Ipp64s
64f	double	Ipp64f
64fc	complex double	Ipp64fc

For functions that operate on a single data type, the *datatype* field contains only one of the values listed above.

If a function operates on source and destination signals that have different data types, the respective data type identifiers are listed in the function name in order of source and destination as follows:

`<datatype> = <src1Depth>[src2Depth][dstDepth]`

For example, the function `ippsDotProd_16s16sc_Sfs` computes the dot product of 16-bit short and 16-bit complex short source vectors and stores the result in a 16-bit complex short destination vector. The *dstDepth* modifier is not present in the name because the second operand and the result are of the same type. The result is scaled and saturated.

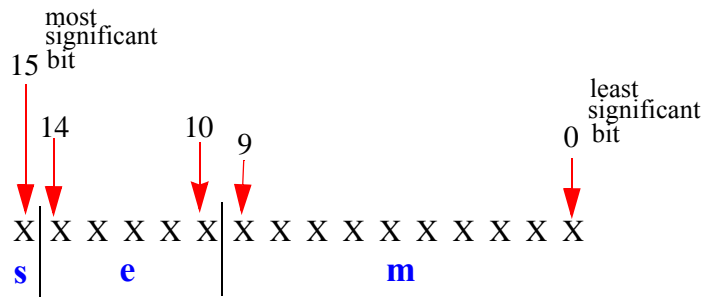
There are several data types, namely 24u, 24s and 16f, that are not supported by the Intel IPP but can be readily converted to supported data types for further processing by the library functions.

For the unsigned 24u data, each vector element consists of three consecutive bytes represented as Ipp8u data types. It has a little-endian byte order when a lower order byte is at the lower address. These data may be converted to and from 32u or 32f data types by using the appropriate flavors of the Intel IPP function `ippsConvert`.

For the signed 24s data, each vector element consists of three consecutive bytes represented as `ipp8u` data types. It has a little-endian byte order when a lower order byte is at the lower address. The sign is represented by the most significant bit of the highest order byte. These data may be converted to and from 32s or 32f data types by using the appropriate flavors of the Intel IPP function `ippsConvert`.

For the 16f format, 16-bit floating point data (half type) can represent positive and negative numbers, whose magnitude is between roughly $6.1e^{-5}$ and $6.5e^4$, with a relative error of $9.8e^{-4}$; numbers smaller than $6.1e^{-5}$ can be represented with an absolute error of $6.0e^{-8}$. All integers from -2048 to $+2048$ can be represented exactly.

The figure below illustrates the bit-layout for a half number:



s is the sign-bit, **e** is the exponent, and **m** is the significand.

These data may be converted to and from 16s and 32f data types by using the appropriate flavors of the Intel IPP function `ippsConvert`.

Descriptor

The *descriptor* field further describes the data associated with the operation. It may contain implied parameters and/or indicate additional required parameters. To minimize the number of code branches in the function and thus reduce potentially unnecessary execution overhead, most of the general functions were split into separate primitive functions, with some of their parameters entering the primitive function name as descriptors.

However, some functions may still have parameters that determine internal operations (for example, `ippsThreshold`).

The following descriptors are used in signal processing functions:

- I Operation is in-place (default is not-in-place).

<code>Sfs</code>	Saturation and fixed scaling mode (default is saturation and no scaling).
<code>Dx</code>	Signal is x-dimensional (default is <code>D1</code>).
<code>L</code>	One pointer is used for each row (in <code>D2</code> case).

The abbreviations of descriptors in function names are always presented in alphabetical order.

Not all functions have every abbreviation listed above. For example, in-place mode makes no sense for a copy operation.

Arguments

The *arguments* field specifies the function arguments.

The order of arguments is as follows:

1. All source operands. Constants follow vectors.
2. All destination operands. Constants follow vectors.
3. Other, operation-specific arguments.

The argument name have the following conventions:

- All arguments defined as pointers start with *p*, for example, *pPhase*, *pSrc*, *pSeed*; and arguments defined as values start with a lowercase letter, for example, *val*, *src*, *srcLen*.
- Each new part of a argument name starts with an uppercase character, without underscore; for example, *pSrc*, *lenSrc*, *pDlyLine*.
- Each argument name specifies its functionality. Source arguments are named *pSrc* or *src*, sometimes followed by names or numbers, e.g., *pSrc2*, *srcLen*. Output arguments are named *pDst* or *dst* followed by names or numbers, e.g., *pDst*, *dstLen*. For in-place operations, the input/ output argument contains the name *pSrcDst*.

Structures and Enumerators

This section describes the structures and enumerators used by the Intel IPP for signal processing.

Library Version Structure

The `IppLibraryVersion` structure describes the current Intel IPP software version. The main fields of this structure are:

- integer fields *major* and *minor*, which store version numbers;

- string field *Name* storing the Intel IPP version name, e.g., “ippsa6”;
- string field *Version*, storing the version description, e.g., “v0.0 Alpha 0.0.3.3”.

Complex Data Structures

Complex numbers in Intel IPP are described by the structures, containing two numbers of the respective data type which are real and imaginary parts of the complex number.

For example, the single precision complex number is described by the `Ipp32fc` structure as follows:

```
typedef struct {  
    Ipp32f re;  
    Ipp32f im;  
} Ipp32fc;
```

The following complex data types are defined: `Ipp16sc`, `Ipp32fc`, `Ipp64fc`.

Function Context Structures

Some Intel IPP functions that perform such operations as filtering, Fourier and wavelet transforms, use context structures to store function-specific information.

For example, the `IppsFFTSpec` structure stores twiddle factors and bit reverse indexes needed in the fast Fourier transform.

Two different kinds of structures are used: structures with `Spec` suffix in the name, which are not modified during function operation, and structures with `State` suffix in the name, which are modified during operation.

These context-related structures are not defined in public headers, and you can not access the structure fields. It was done because the function context interpretation is processor dependent. Thus, you may only use context-related functions and may not create a function context as an automatic variable.

Enumerators

The `IppStatus` constant enumerates the status values returned by the Intel IPP functions, indicating whether the operation was error-free or not. See section [Error Reporting](#) in this chapter for more information on the set of valid status values and corresponding error messages for signal processing functions.

The `IppCmpOp` enumeration defines the type of relational operator to be used by [threshold functions](#):

```
typedef enum {
    ippCmpLess,
    ippCmpLessEq,
    ippCmpEq,
    ippCmpGreaterEq,
    ippCmpGreater
} IppCmpOp;
```

The `IppRoundMode` enumeration defines the rounding mode to be used by [conversion functions](#):

```
typedef enum {
    ippRndZero,
    ippRndNear
} IppRoundMode;
```

The `IppHintAlgorithm` enumeration defines the type of code to be used in some transform operations, i. e. faster but less accurate, or vice-versa, more accurate but slower. For more information on using this enumerator, see [“Flag and Hint Arguments”](#).

```
typedef enum {
    ippAlgHintNone,
    ippAlgHintFast,
    ippAlgHintAccurate
} IppHintAlgorithm;
```

The `IppCpuType` enumerates processor types returned by the [ippGetCpuType](#) function:

```
typedef enum {
    /* Enumeration: Processor: */
    ippCpuUnknown = 0x0,
    ippCpuPP, /* Intel(R) Pentium(R) processor */
    ippCpuPMX, /* Pentium(R) processor
               with MMX(TM) technology */
    ippCpuPPR, /* Pentium(R) Pro processor */
    ippCpuPII, /* Pentium(R) II processor */
    ippCpuPIII, /* Pentium(R) III processor
                and Pentium(R)III Xeon(R) processor */
    ippCpuP4, /* Pentium(R) 4 processor
               and Intel(R) Xeon(R) processor */
    ippCpuP4HT, /* Pentium(R) 4 processor with HT Technology */
    ippCpuP4HT2, /* Pentium(R) 4 processor with
                  Streaming SIMD Extensions 3 */
    ippCpuCentrino, /* Intel(R) Centrino(TM) mobile technology */
    ippCpuITP = 0x10, /* Intel(R) Itanium(R) processor */
}
```



```

    ippCpuITP2          /* Itanium(R) 2 processor          */
    ippCpuEM64T = 0x20 /* IA-32 processor with Intel(R) Extended Memory 64 Technology */
    ippCpuSSE   = 0x40 /* Processor supports Pentium(R) III processor instruction set */
    ippCpuSSE2  /* Processor supports Streaming SIMD Extensions 2 instruction set */
    ippCpuSSE3  /* Processor supports Streaming SIMD Extensions 3 instruction set */
    ippCpuX866  /* Processor supports 64 bit extension */
} IppCpuType;

```

The `IppWinType` enumeration defines the type of window to be used by the [FIR Filter Coefficient Generating Functions](#):

```

typedef enum {
    ippWinBartlett,
    ippWinBlackman,
    ippWinHamming,
    ippWinHann,
    ippWinRect
} IppWinType;

```

The `IppLZ77ComprLevel` enumeration defines the compression level to be used by the [ZLIB data compression functions](#):

```

typedef enum {
    IppLZ77FastCompr,
    IppLZ77AverageCompr,
    IppLZ77BestCompr
} IppLZ77ComprLevel;

```

The `IppLZ77Chcksm` enumeration defines what algorithm is used to compute the checksum by the [ZLIB data compression functions](#):

```

typedef enum {
    IppLZ77NoChcksm,
    IppLZ77Adler32,
    IppLZ77CRC32
} IppLZ77Chcksm;

```

The `IppLZ77Flush` enumeration defines what encoding mode is used by the [ZLIB data compression functions](#):

```

typedef enum {
    IppLZ77NoFlush,
    IppLZ77SyncFlush,

```

```
        IppLZ77FullFlush,  
        IppLZ77FinishFlush  
    } IppLZ77Flush;
```

The `IppLZ77DeflateStatus` enumeration defines the encoding status that is used by the [ZLIB data compression functions](#):

```
typedef enum {  
    IppLZ77StatusInit,  
    IppLZ77StatusLZ77Process,  
    IppLZ77StatusHuffProcess,  
    IppLZ77StatusFinal  
} IppLZ77DeflateStatus;
```

The `IppLZ77InflateStatus` enumeration defines the decoding status that is used by the [ZLIB data compression functions](#):

```
typedef enum {  
    IppLZ77InflateStatusInit,  
    IppLZ77InflateStatusHuffProcess,  
    IppLZ77InflateStatusLZ77Process,  
    IppLZ77InflateStatusFinal  
} IppLZ77InflateStatus;
```

The `IppLZ77HuffMode` enumeration defines the encoding mode that is used by the [ZLIB data compression functions](#):

```
typedef enum {  
    IppLZ77UseFixed,  
    IppLZ77UseDynamic,  
    IppLZ77UsedStored,  
} IppLZ77HuffMode;
```

The `IppGITStrategyHint` enumeration defines which strategy of encoding is used in some operations by the [GIT data compression functions](#):

```
typedef enum {  
    ippGITNoStrategy,  
    ippGITLeftReorder,  
    ippGITRightReorder,  
    ippGITFixedOrder  
} IppGITStrategyHint;
```

Data Ranges

The range of values that can be represented by each data type lies between the lower and upper bounds. The following table lists data ranges and constant identifiers used in Intel IPP to denote the respective range bounds:

Table 2-2 Data Types and Ranges

Data Type	Lower Bound		Upper Bound	
	Identifier	Value	Identifier	Value
8s	IPP_MIN_8S	-128	IPP_MAX_8S	127
8u		0	IPP_MAX_8U	255
16s	IPP_MIN_16S	-32768	IPP_MAX_16S	32767
16u		0	IPP_MAX_16U	65535
32s	IPP_MIN_32S	-2 ³¹	IPP_MAX_32S	2 ³¹ -1
32u		0	IPP_MAX_32U	2 ³² -1
32f †	IPP_MINABS_32F	1.175494351e ⁻³⁸	IPP_MAXABS_32F	3.402823466e ³⁸
64s	IPP_MIN_64S	-2 ⁶³	IPP_MAX_64S	2 ⁶³ -1
64f †	IPP_MINABS_64F	2.2250738585072014e ⁻³⁰⁸	IPP_MAXABS_64F	1.7976931348623158e ³⁰⁸

† The range for absolute values

Data Alignment

The Intel IPP is built using the compiler option `/Zp16`, which aligns the structure fields on the field size or 16 bytes if the size is greater than 16.

The Intel IPP allows also to align the allocated memory pointer on 32 bytes by means of using the function `ippsMalloc`.

Integer Scaling

Some signal processing functions operating on integer data use scaling of the internally computed output results by the integer *scaleFactor*, which is specified as one of the function arguments. These functions have the *Sfs* descriptor in their names.

The scale factor can be negative, positive, or zero. Scaling is applied because internal computations are generally performed with a higher precision than the data types used for input and output signals.



NOTE. The result of integer operations is always saturated to the destination data type range even when scaling is used.

Scaling of an integer result is done by multiplying the output vector values by $2^{-scaleFactor}$ before the function returns. This helps retain either the output data range or its precision. Usually the scaling with a positive factor is performed by the shift operation. The result is rounded off to the nearest integer number.

For example, the integer `Ipp16s` result of the square operation `ippsSqr` for the input value 200 is equal to 32767 instead of 40000, that is, the result is saturated and the exact value can not be restored.

The scaling of the output value with the factor `scaleFactor = 1` yields the result 20000 which is not saturated, and the exact value can be restored as $20000 * 2$. Thus, the output data range is retained.

The following example shows how the precision can be partially retained by means of scaling.

The integer square root operation `ippsSqrt` (without scaling) for the input value 2 gives the result equal to 1 instead of 1.414. Scaling of the internally computed output value with the factor `scaleFactor = -3` gives the result 11, and permits to restore the more precise value as $11 * 2^{-3} = 1.375$.

Error Reporting

The Intel IPP functions return the status of the performed operation to report errors and warnings to the calling program. Thus, it is up to the application to perform error-related actions and/or recover from the error. The last value of the error status is not stored, and the user is to decide whether to check it or not as the function returns. The status values are of `IppStatus` type and are global constant integers.

[Table 2-3](#) lists status codes and corresponding messages reported by the Intel IPP for signal processing.

Table 2-3 Error Status Values and Messages

Status	Message
ippStsCpuNotSupportedErr	The target cpu is not supported
ippStsMP3FrameHeaderErr	Error in fields IppMP3FrameHeader structure
ippStsMaxLenHuffCodeErr	Huff: Max length of Huffman code is more than expected one
ippStsCodeLenTableErr	Huff: Invalid codeLenTable
ippStsFreqTableErr	Huff: Invalid freqTable
ppStsRegExpOptionsErr	RegExp: Options for pattern are incorrect
ppStsRegExpErr	RegExp: The structure pRegExpState contains wrong data
ippStsRegExpMatchLimitErr	RegExp: The match limit has been exhausted
ippStsRegExpQuantifierErr	RegExp: wrong quantifier
ippStsRegExpGroupingErr	RegExp: wrong grouping
ippStsRegExpBackRefErr	RegExp: wrong back reference
ippStsRegExpChClassErr	RegExp: wrong character class
ippStsRegExpMetaChErr	RegExp: wrong metacharacter
ippStsMP3SideInfoErr	Error in fields IppMP3SideInfo structure
ippStsAacPrgNumErr	AAC: Invalid number of elements for one program
ippStsAacSectCbErr	AAC: Invalid section codebook
ippStsAacSfValErr	AAC: Invalid scalefactor value
ippStsAacCoefValErr	AAC: Invalid quantized coefficient value
ippStsAacMaxSfbErr	AAC: Invalid coefficient index
ippStsAacPredSfbErr	AAC: Invalid predicted coefficient index
ippStsAacPlsDataErr	AAC: Invalid pulse data attributes
ippStsAacGainCtrErr	AAC: Gain control not supported
ippStsAacSectErr	AAC: Invalid number of sections
ippStsAacTnsNumFiltErr	AAC: Invalid number of TNS filters
ippStsAacTnsLenErr	AAC: Invalid TNS region length
ippStsAacTnsOrderErr	AAC: Invalid order of TNS filter
ippStsAacTnsCoefResErr	AAC: Invalid bit-resolution for TNS filter coefficients
ippStsAacTnsCoefErr	AAC: Invalid TNS filter coefficients

Table 2-3 Error Status Values and Messages (continued)

<code>ippStsAacTnsDirectErr</code>	AAC: Invalid TNS filter direction
<code>ippStsAacTnsProfileErr</code>	AAC: Invalid TNS profile
<code>ippStsAacErr</code>	AAC: Internal error
<code>ippStsAacBitOffsetErr</code>	AAC: Invalid current bit offset in bitstream
<code>ippStsAacAdtsSyncWordErr</code>	AAC: Invalid ADTS syncword
<code>ippStsAacSmplRateIdxErr</code>	AAC: Invalid sample rate index
<code>ippStsAacWinLenErr</code>	AAC: Invalid window length (not short or long)
<code>ippStsAacWinGrpErr</code>	AAC: Invalid number of groups for current window length
<code>ippStsAacWinSeqErr</code>	AAC: Invalid window sequence range
<code>ippStsAacComWinErr</code>	AAC: Invalid common window flag
<code>ippStsAacStereoMaskErr</code>	AAC: Invalid stereo mask
<code>ippStsAacChanErr</code>	AAC: Invalid channel number
<code>ippStsAacMonoStereoErr</code>	AAC: Invalid mono-stereo flag
<code>ippStsAacStereoLayerErr</code>	AAC: Invalid this Stereo Layer flag
<code>ippStsAacMonoLayerErr</code>	AAC: Invalid this Mono Layer flag
<code>ippStsAacScalableErr</code>	AAC: Invalid scalable object flag
<code>ippStsAacObjTypeErr</code>	AAC: Invalid audio object type
<code>ippStsAacWinShapeErr</code>	AAC: Invalid window shape
<code>ippStsAacPcmModeErr</code>	AAC: Invalid PCM output interleaving indicator
<code>ippStsVLCUsrTblHeaderErr</code>	VLC: Invalid header inside table
<code>ippStsVLCUsrTblUnsupportedFmtErr</code>	VLC: Unsupported table format
<code>ippStsVLCUsrTblEscAlgTypeErr</code>	VLC: Unsupported Ecs-algorithm
<code>ippStsVLCUsrTblEscCodeLengthErr</code>	VLC: Incorrect Esc-code length inside table header
<code>ippStsVLCUsrTblCodeLengthErr</code>	VLC: Unsupported code length inside table
<code>ippStsVLCInternalTblErr</code>	VLC: Invalid internal table
<code>ippStsVLCInputDataErr</code>	VLC: Invalid input data
<code>ippStsVLCAACEscCodeLengthErr</code>	VLC: Invalid AAC-Esc code length
<code>ippStsIncorrectLSPErr</code>	Incorrect Linear Spectral Pair values
<code>ippStsNoRootFoundErr</code>	No roots are found for equation
<code>ippStsLengthErr</code>	Wrong value of string length
<code>ippStsFBankFreqErr</code>	Incorrect value of the filter bank frequency parameter

Table 2-3 Error Status Values and Messages (continued)

ippStsFBankFlagErr	Incorrect value of the filter bank parameter
ippStsFBankErr	Filter bank is not correctly initialized
ippStsNegOccErr	Negative occupation count
ippStsCdbkFlagErr	Incorrect value of the codebook flag parameter
ippStsSVDConvErr	No convergence of SVD algorithm
ippStsToneMagnErr	Tone magnitude is less than or equal to zero
ippStsToneFreqErr	Tone frequency is negative, or greater than or equal to 0.5
ippStsTonePhaseErr	Tone phase is negative, or greater than or equal to 2π
ippStsTrnglMagnErr	Triangle magnitude is less than or equal to zero
ippStsTrnglFreqErr	Triangle frequency is negative, or greater than or equal to 0.5
ippStsTrnglPhaseErr	Triangle phase is negative, or greater than or equal to 2π
ippStsTrnglAsymErr	Triangle asymmetry is less than $-\pi$, or greater than or equal to π
ippStsHugeWinErr	Incorrect size of the Kaiser window
ippStsJaehneErr	Magnitude value is negative
ippStsStepErr	Step value is not valid
ippStsDlyLineIndexErr	Invalid value of the delay line sample index
ippStsStrideErr	Stride value is less than the row length
ippStsEpsValErr	Negative epsilon value error
ippStsScaleRangeErr	Scale bounds are out of range
ippStsThresholdErr	Invalid threshold bounds
ippStsWtOffsetErr	Invalid offset value of the wavelet filter
ippStsAnchorErr	Anchor point is outside the mask
ippStsMaskSizeErr	Invalid mask size
ippStsShiftErr	Shift value is less than zero
ippStsSampleFactorErr	Sampling factor is less than or equal to zero
ippStsSamplePhaseErr	Phase value is out of range, $0 \leq \text{phase} < \text{factor}$
ippStsFIRMRFactorErr	MR FIR sampling factor is less than or equal to zero
ippStsFIRMRPhaseErr	MR FIR sampling phase parameter is negative, or greater than or equal to the sampling factor
ippStsRelFreqErr	Relative frequency value is out of range
ippStsFIRLenErr	Length of a FIR filter is less than or equal to zero
ippStsIIROrderErr	Order of an IIR filter is less than or equal to zero

Table 2-3 Error Status Values and Messages (continued)

<code>ippStsResizeFactorErr</code>	Resize factor(s) is less than or equal to zero
<code>ippStsDivByZeroErr</code>	An attempt to divide by zero
<code>ippStsInterpolationErr</code>	Invalid interpolation mode
<code>ippStsMirrorFlipErr</code>	Invalid flip mode
<code>ippStsMoment00ZeroErr</code>	Moment value $M(0,0)$ is too small to continue calculations
<code>ippStsThreshNegLevelErr</code>	Negative value of the level in the threshold operation
<code>ippStsContextMatchErr</code>	Context parameter doesn't match the operation
<code>ippStsFftFlagErr</code>	Invalid value of the FFT flag parameter
<code>ippStsFftOrderErr</code>	Invalid value of the FFT order parameter
<code>ippStsMemAllocErr</code>	Not enough memory allocated for the operation
<code>ippStsNullPtrErr</code>	Null pointer error
<code>ippStsSizeErr</code>	Wrong value of the data size
<code>ippStsBadArgErr</code>	Function argument/parameter is bad
<code>ippStsErr</code>	Unknown/unspecified error
<code>ippStsNoErr</code>	No error, it's OK
<code>ippStsNoOperation</code>	No operation has been executed
<code>ippStsMisalignedBuf</code>	Misaligned pointer in operation in which it must be aligned
<code>ippStsSqrtNegArg</code>	Negative value(s) of the argument in the function <code>Sqrt</code>
<code>ippStsInvByZero</code>	Inf result. Zero value was met by <code>InvThresh</code> with zero level
<code>ippStsEvenMedianMaskSize</code>	Even size of the Median Filter mask was replaced by the odd one
<code>ippStsDivByZero</code>	Zero value(s) of the divisor in the function <code>Div</code>
<code>ippStsLnZeroArg</code>	Zero value(s) of the argument in the function <code>Ln</code>
<code>ippStsLnNegArg</code>	Negative value(s) of the argument in the function <code>Ln</code>
<code>ippStsNanArg</code>	Not a Number (NaN) argument value warning
<code>ippStsResFloor</code>	All result values are floored
<code>ippStsOverflow</code>	Overflow occurred in the operation
<code>ippStsZeroOcc</code>	Zero occupation count
<code>ippStsUnderflow</code>	Underflow occurred in the operation
<code>ippStsSingularity</code>	Singularity occurred in the operation
<code>ippStsDomain</code>	Argument is out of the function domain
<code>ippStsNotIntelCpu</code>	The target cpu is not Genuine Intel

Table 2-3 Error Status Values and Messages (continued)

<code>ippStsCpuMismatch</code>	The library for given cpu cannot be set
<code>ippStsNotIppFunctionFound</code>	Application does not contain IPP functions calls
<code>ippStsDllNotFoundBestUsed</code>	The newest version of IPP DLL's not found by dispatcher
<code>ippStsNoOperationInDll</code>	The function does nothing in the dynamic version of the library
<code>ippStsOvermuchStrings</code>	Number of destination strings is more than expected
<code>ippStsOverlongString</code>	Length of one of the destination strings is more than expected
<code>ippStsSrcSizeLessExpected</code>	DC: The size of source buffer is less than expected one
<code>ippStsDstSizeLessExpected</code>	DC: The size of destination buffer is less than expected one
<code>ippStsNotSupportedCpu</code>	The CPU is not supported
<code>ippStsUnkhownCacheSize</code>	The CPU is supported, but the size of the cache is unknown.

The status codes ending with `Err` (except for the `ippStsNoErr` status) indicate an error; the integer values of these codes are negative. When an error occurs, the function execution is interrupted.

All other status codes indicate warnings. When a specific case is encountered, the function execution is completed and the corresponding warning status is returned.

For example, if the integer function `ippsDiv_8u` meets an attempt to divide a positive value by zero, the function execution is not aborted. The result of the operation is set to the maximum value that can be represented by the source data type, and the function returns the warning status `ippStsDivByZero`. This is the case for the vector-vector operation `ippsDiv`. For the vector-scalar division operation `ippsDivC`, the function behavior is different: if the constant divisor is zero, then the function stops execution and returns immediately with the error status `ippStsDivByZeroErr`.

Code Examples

The manual contains a number of code examples that use Intel IPP functions and serve to demonstrate both some particular features of the primitives and how the primitives can be called. Many of these code examples output result data together with status code and associated messages in case when error or warning condition was met.

To keep the example code simpler, special definitions of print statements are used that get output strings look exactly the way is needed for better representation of both real and complex results of different format, as well as print status codes and messages.

The code definitions given below make it possible to build the examples contained in the manual by straightforward copying and pasting the example code fragments.

```

/// the functions providing simple output of the result
/// they are for real and complex data

#define genPRINT(TYPE,FMT) \
void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus st ) { \
    int n; \
    if( st > ippStsNoErr ) \
        printf( "-- warning %d, %s\n", st, ippGetStatusString( st )); \
    else if( st < ippStsNoErr ) \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    printf(" %s ", msg ); \
    for( n=0; n<len; ++n ) printf( FMT, buf[n] ); \
    printf("\n" ); \
}

#define genPRINTcplx(TYPE,FMT) \
void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus st ) { \
    int n; \
    if( st > ippStsNoErr ) \
        printf( "-- warning %d, %s\n", st, ippGetStatusString( st )); \
    else if( st < ippStsNoErr ) \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    printf(" %s ", msg ); \
    for( n=0; n<len; ++n ) printf( FMT, buf[n].re, buf[n].im ); \
    printf("\n" ); \
}

genPRINT( 64f, " %f" )
genPRINT( 32f, " %f" )
genPRINT( 16s, " %d" )

genPRINTcplx( 64fc, " {%f,%f}" )
genPRINTcplx( 32fc, " {%f,%f}" )
genPRINTcplx( 16sc, " {%d,%d}" )

```

Support Functions

3

This chapter describes Intel[®] IPP support functions that are used to:

- retrieve information about the current Intel IPP software version
- allocate and free memory that is needed for the operation of other Intel IPP functions.
- perform specific auxiliary operations

The full list of these functions is given in [Table 3-1](#).

Table 3-1 Intel IPP Support Functions

Function Base Name	Operation
Version Information Functions	
ippGetLibVersion	Returns information about the active library version.
Memory Allocation Functions	
ippMalloc	Allocates memory aligned to 32-byte boundary.
ippFree	Frees memory allocated by the function <code>ippMalloc</code> .
Common Functions	
ippGetStatusString	Translates a status code into a message.
ippGetCpuType	Returns a processor type.
ippGetCpuClocks	Returns a current value of the time stamp counter (TSC) register.
ippGetCpuFreqMhz	Estimates the processor frequency.
ippGetMaxCacheSizeB	
ippSetFlushToZero	Enables or disables flush-to-zero mode.
ippSetDenormAreZeros	Enables or disables denormals-are-zero mode.
ippAlignPtr	Aligns a pointer to the specified number of bytes.

Table 3-1 Intel IPP Support Functions (continued)

Function Base Name	Operation
<code>ippSetNumThreads</code>	Sets the number of threads in the multithread environment.
<code>ippGetNumThreads</code>	Returns the number of existing threadsin the multithread environment.
<code>ippMalloc</code>	Allocates memory aligned to 32-byte boundary.
<code>ippFree</code>	Frees memory allocated by the function <code>ippMalloc</code> .
Dispatcher Control Functions	
<code>ippStaticInit</code>	Automatically selects the most appropriate static code.
<code>ippStaticInitCpu</code>	Initializes the specified version of the static code.

Version Information Functions

These functions return the version number and other information about the active Intel IPP software.

ippsGetLibVersion

Returns information about the active version of Intel IPP signal processing software.

```
const IppLibraryVersion* ippsGetLibVersion(void);
```

Description

The function `ippsGetLibVersion` is declared in the `ipps.h` file. This function returns a pointer to a static data structure `IppLibraryVersion` that contains information about the current version of the Intel IPP software for signal processing. There is no need for you to release memory referenced by the returned pointer, as it points to a static variable. The following fields of the `IppLibraryVersion` structure are available:

<i>major</i>	is the major number of the current library version.
<i>minor</i>	is the minor number of the current library version.
<i>Name</i>	is the name of the current library version.
<i>Version</i>	is the library version string.
<i>BuildDate</i>	is the library version actual build date.

For example, if the library version is “v1.2 Beta”, library name is “ippsm6”, and build date is “Jul 20 99”, then the fields in this structure are set as:

```
major = 1, minor = 2, Name = “ippsm6”,
Version = “v1.2 Beta”, BuildDate = “Jul 20 99”
```

[Example 3-1](#) shows how to use the function `ippsGetLibVersion`.

Example 3-1 Using the `ippsGetLibVersion` Function

```
void libinfo(void) {
    const IppLibraryVersion* lib = ippsGetLibVersion();
    printf(“%s %s %d.%d.%d.%d\n”, lib->Name, lib->Version,
        lib->major, lib->minor, lib->majorBuild, lib->build);
}
```

Output:

```
ippsa6 v0.0 Alpha 0.0.5.5
```



NOTE. Each sub-library that is used in the signal processing domain has its own similar function to retrieve information about the active library version. These functions are: `ippGetLibVersion`, `ippacGetLibVersion`, `ippchGetLibVersion`, `ippdcGetLibVersion`, `ippscGetLibVersion`, `ippsrGetLibVersion`, `ippvmGetLibVersion`. They are declared in the following header files: `ippcore.h`, `ippac.h`, `ippch.h`, `ippdc.h`, `ippsc.h`, `ippsr.h`, `ippvm.h`, respectively, and have the same interface as the above described function.

Memory Allocation Functions

This section describes the Intel IPP signal processing functions that allocate aligned memory blocks for data of required type or free the previously allocated memory. The size of allocated memory is specified by the number of elements allocated, *len*.



NOTE. The only function to free the memory allocated by any of these functions is `ippsFree()`.

ippsMalloc

Allocates memory aligned to 32-byte boundary.

Syntax

```
Ipp8u* ippsMalloc_8u(int len);
Ipp16u* ippsMalloc_16u(int len);
Ipp32u* ippsMalloc_32u(int len);
Ipp8s* ippsMalloc_8s(int len);
Ipp16s* ippsMalloc_16s(int len);
Ipp32s* ippsMalloc_32s(int len);
Ipp64s* ippsMalloc_64s(int len);
Ipp32f* ippsMalloc_32f(int len);
Ipp64f* ippsMalloc_64f(int len);
Ipp8sc* ippsMalloc_8sc(int len);
Ipp16sc* ippsMalloc_16sc(int len);
Ipp32sc* ippsMalloc_32sc(int len);
Ipp64sc* ippsMalloc_64sc(int len);
Ipp32fc* ippsMalloc_32fc(int len);
Ipp64fc* ippsMalloc_64fc(int len);
```

Parameters

len Number of elements to allocate.

Description

The function `ippsMalloc` is declared in the `ipps.h` file. This function allocates memory block aligned to a 32-byte boundary for elements of different data types.

Return Values

The return value of `ippsMalloc` is a pointer to an aligned memory block. If no memory is available in the system, then the `NULL` value is returned. To free this block, use the function `ippsFree`.

ippsFree

Frees memory allocated by the function `ippsMalloc`.

Syntax

```
void ippsFree(void* ptr);
```

Parameters

ptr Pointer to a memory block to be freed. The memory block pointed to with *ptr* has been allocated by the function `ippsMalloc`.

Description

The function `ippsFree` is declared in the `ipps.h` file. This function frees the aligned memory block allocated by the function `ippsMalloc`.



CAUTION. You can not use the function `ippsFree` to free memory allocated by standard functions like `malloc` or `calloc`, nor can the memory allocated by `ippsMalloc` be freed by `free`.

Common Functions

This section describes the Intel IPP functions that perform special operations common for all domains. For example, the function `ippGetStatusString` gets a brief description of the status code returned by the current Intel IPP software. The function `ippGetCpuType` retrieves the type of the processor in your system. The function `ippGetCpuClocks` gets the current number of processor clocks, which is widely used for operations timing. The functions `ippSetFlushToZero` and `ippSetDenormAreZeros` enable or disable special processor modes. The function `ippAlignPtr` performs pointer alignment. The functions `ippMalloc` and `ippFree` allow to allocate and free the memory block aligned to 32-byte boundary. The specific subset contains functions to control the dispatching of the merged static libraries. All these functions are grouped in the separate sub-library called `ippcore`.

ippGetStatusString

Translates a status code into a message.

Syntax

```
const char* ippGetStatusString(IppStatus StsCode);
```

Parameters

StsCode Code that indicates the status type (see [Table 2-3](#)).

Description

The function `ippGetStatusString` is declared in the `ippcore.h` file. This function returns a pointer to the text string associated with a status code of type `IppStatus`. Use this function to produce error and warning messages for users. The returned pointer is a pointer to an internal static buffer and need not be released.

A code example below shows how to use the function `ippGetStatusString`. If you call an Intel IPP function, in this case, `ippsAddC_16s_I` with a NULL pointer, it returns an error code -8. The status information function translates this code into the corresponding message “Null Pointer Error”.

Example 3-2 Using the `ippGetStatusString` Function

```
void statusinfo(void) {  
    IppStatus st = ippsAddC_16s_I (3, 0, 0);  
    printf("%d : %s\n", st, ippGetStatusString(st));  
}
```

Output:
-8, Null Pointer Error

`ippGetCpuType`

Returns a processor type.

Syntax

```
IppCpuType ippGetCpuType (void);
```

Description

The function `ippGetCpuType` is declared in the `ippcore.h` file. This function detects the processor type used in your computer system and returns an appropriate `IppCpuType` variable value. [Table 3-2](#) lists possible values and their meaning.

Table 3-2 Processor Types Detected in Intel IPP

Returned Variable Value	Processor Type
<code>ippCpuPP</code>	Intel [®] Pentium [®] processor
<code>ippCpuPMX</code>	Intel [®] Pentium [®] processor with MMX™ technology
<code>ippCpuPPR</code>	Intel [®] Pentium [®] Pro processor
<code>ippCpuPII</code>	Intel [®] Pentium [®] II processor

Table 3-2 Processor Types Detected in Intel IPP (continued)

Returned Variable Value	Processor Type
ippCpuPIII	Intel® Pentium® III processor or Intel® Pentium® III Xeon® processor
ippCpuP4	Intel® Pentium® 4 processor or Intel® Xeon® processor
ippCpuP4HT	Intel Pentium 4 processor with Hyper-Threading Technology
ippCpuP4HT2	Intel Pentium 4 Processor with Streaming SIMD Extensions 3
ippCpuCentrino	Intel® Centrino™ mobile technology
ippCpuITP	Intel® Itanium® processor
ippCpuITP2	Intel® Itanium® 2 processor
ippCpuEM64T	IA-32 processor with Intel® Extended Memory 64 Technology
ippCpuSSE	Processor supports Intel Pentium III processor instruction set
ippCpuSSE2	Processor with Streaming SIMD Extensions 2 instruction set
ippCpuSSE3	Processor with Streaming SIMD Extensions 3 instruction set
ippCpuX8664	Processor supports 64 bit extension
ippCpuUnknown	Unknown processor

ippGetCpuClocks

Returns a current value of the time stamp counter (TSC) register.

Syntax

```
Ipp64u ippGetCpuClocks (void);
```

Description

The function `ippGetCpuClocks` is declared in the `ippcore.h` file. This function reads the current state of the time stamp counter (TSC) register and returns its value. A hardware exception is possible if TSC reading is not supported by the current chipset.

ippGetCpuFreqMhz

Estimates the processor operating frequency.

Syntax

```
IppStatus ippGetCpuFreqMhz(int* pMhz);
```

Parameters

pMhz Pointer to the output result.

Description

The function `ippGetCpuFreqMhz` is declared in the `ippcore.h` file. This function estimates the processor operating frequency and returns its value in MHz as an integer stored in *pMhz*.

Note that no exact value of frequency is guaranteed so that the estimated value can vary depending on the processor workload.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error condition when the *pMhz* pointer is NULL.

ippGetMaxCacheSizeB

Returns maximum size of the L2 and L3 caches of the processor.

Syntax

```
IppStatus ippGetMaxCacheSizeB(int* pSizeByte)
```

Parameters

pSizeByte Pointer to the output result.

Description

The function `ippGetMaxCacheSizeB` is declared in the `ippcore.h` file. This function finds the maximum size (in bytes) of the L2 and L3 caches of the processor used in your computer system. The result is stored in the `pSizeByte`.

Note the function supports only Intel® IA-32 processors including those with Intel® EM64T. If the processor is not supported, or size of cache is unknown, the result is 0, and the function returns corresponding warning message.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition when the <code>pSizeByte</code> pointer is NULL.
<code>ippStsNotSupportedCpu</code>	Indicates that processor is not supported.
<code>ippStsUnknownCacheSize</code>	Indicates that size of the cache is unknown.

ippSetFlushToZero

Enables or disables flush-to-zero mode.

Syntax

```
IppStatus ippSetFlushToZero(int value, unsigned int* pUMask);
```

Parameters

<code>value</code>	Switch to set or clear the corresponding bit of the MXCSR register.
<code>pUMask</code>	Pointer to the current underflow exception mask; may be set to NULL.

Description

The function `ippSetFlushToZero` is declared in the `ippcore.h` file. This function enables (when the `value` is not equal to 0) or disables (when the `value` is equal to 0) a flush-to-zero (FTZ) mode of processors that support Streaming SIMD Extensions (SSE) instructions. The FTZ mode controls the masked response to a SIMD floating-point underflow condition. The FTZ mode

is provided primarily for performance reasons. At the cost of a slight precision loss, FTZ mode enables faster execution of applications where underflows are common and rounding the underflow result to zero can be tolerated.

FTZ mode is possible only when the mask register is in a certain state. The `ippSetFlushToZero` function checks and changes this state if necessary. After disabling the FTZ mode, you can restore the initial mask register state. To do this, you must declare a variable of `unsigned integer` type in your application and point to it in the `pUMask` parameter of the `ippSetFlushToZero` function. The initial state of mask register will be saved in this location and can be restored later. If you do not need to restore the initial mask state, then the `pUMask` pointer may be set to `NULL`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsCpuNotSupportedErr</code>	Indicates an error condition when the FTZ mode is not supported by the processor.

ippSetDenormAreZeros

Enables or disables denormals-are-zero mode.

Syntax

```
IppStatus ippSetDenormAreZeros(int value);
```

Parameters

<code>value</code>	Switch to set or clear the corresponding bit of the MXCSR register.
--------------------	---

Description

The function `ippSetDenormAreZeros` is declared in the `ippcore.h` file. This function enables (when the `value` is not equal to 0) or disables (when the `value` is equal to 0) the denormals-are-zero (DAZ) mode of processors that support Streaming SIMD Extensions (SSE) instructions. The DAZ mode controls the processor response to a SIMD floating-point denormal operand condition. When the DAZ flag is set, the processor converts all denormal source operands to zero with the sign of the original operand before performing any computations on source data. The DAZ mode is provided to improve processor performance for applications such as streaming media processing, where rounding a denormal operand to zero does not noticeably affect the quality of the processed data.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsCpuNotSupportedErr</code>	Indicates an error condition when the DAZ mode is not supported by the processor.

ippAlignPtr

Aligns a pointer to the specified number of bytes.

Syntax

```
void* ippAlignPtr(void* ptr, int alignBytes);
```

Parameters

<code>ptr</code>	Processor type.
<code>alignBytes</code>	Number of bytes to align. Possible values are the powers of 2, that is, 2, 4, 8, 16 and so on.

Description

The function `ippAlignPtr` is declared in the `ippcore.h` file. This function returns a pointer `ptr` aligned to the specified number of bytes `alignBytes`. Possible values of `alignBytes` are powers of two. The function does not check the validity of this parameter.



CAUTION. Do not free the pointer returned by the function, but free the original pointer.

ippSetNumThreads

Sets the number of threads in the multithreading environment.

Syntax

```
IppStatus ippSetNumThreads(int numThr);
```

Parameters

numThr Number of threads, should be greater than 0.

Description

The function `ippSetNumThreads` is declared in the `ippcore.h` file. This function sets the number of OMP threads. A number of established threads may be less than specified *numThr*.



WARNING. This function operates only in the dynamic version of the library. In static version it returns a warning status `ippStsNoOperation`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsSizeErr</code>	Indicates an error condition if <i>numThr</i> is less than or equal to 0.
<code>ippStsNoOperation</code>	Indicates that there is no such operation in the static version of the library.

ippGetNumThreads

Returns the number of existing threads in the multithreading environment.

Syntax

```
IppStatus ippGetNumThreads(int* pNumThr);
```

Parameters

pNumThr Pointer to the number of threads.

Description

The function `ippGetNumThreads` is declared in the `ippcore.h` file. This function returns the number of OMP threads specified by the user previously. If it has not been specified, the function returns the initial number of threads that depends on the number of logical processors



WARNING. This function operates only in the dynamic version of the library. In static version it returns a warning status `ippStsNoOperation`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition when the <i>pMhz</i> pointer is <code>NULL</code> .
<code>ippStsNoOperation</code>	Indicates that there is no such operation in the static version of the library.

ippMalloc

Allocates memory aligned to 32-byte boundary.

Syntax

```
void* ippMalloc(int length);
```

Parameters

len Number of elements to allocate.

Description

The function `ippMalloc` is declared in the `ippcore.h` file. This function allocates memory block aligned to a 32-byte boundary.

Return Values

The return value of `ippMalloc` is a pointer to an aligned memory block. To free this block, use only the function `ippFree`.

ippFree

Frees memory allocated by the function `ippMalloc`.

Syntax

```
void ippFree(void* ptr);
```

Parameters

ptr Pointer to a memory block to be freed.

Description

The function `ippFree` is declared in the `ippcore.h` file. This function frees the aligned memory block allocated by the function `ippMalloc`.



CAUTION. You can not use the function `ippFree` to free memory allocated by any other functions like `malloc`, nor can the memory allocated by `ippMalloc` be freed by `free`.

Dispatcher Control Functions

This section describes Intel IPP functions that control the dispatchers of the merged static libraries.

ippStaticInit

Automatically selects the most appropriate static code.

Syntax

```
IppStatus ippStaticInit(void);
```

Description

The function `ippStaticInit` is declared in the `ippcore.h` file. This function detects the processor type used in the user computer system and sets the most appropriate processor-specific static code of the Intel IPP software.

The function `ippStaticInit` should not be used in the driver implementation.



WARNING. This function operates only in the static version of the library.



CAUTION. You can not use any other Intel IPP function while the function `ippStaticInit` continues execution.

Return Values

<code>ippStsNoErr</code>	Indicates that the most appropriate static code of the Intel IPP software is successfully set.
<code>ippStsNonIntelCpu</code>	Indicates that the static version of generic code for Intel [®] Architecture is set.
<code>ippStsNoOperationInDll</code>	Indicates that there is no such operation in the dynamic version of the library.

ippStaticInitCpu

Initializes the specified version of the static code.

Syntax

```
IppStatus ippStaticInitCpu(IppCpuType cpu);
```

Parameters

cpu Processor type.

Description

The function `ippStaticInitCpu` is declared in the `ippcore.h` file. This function sets the processor-specific static code of the Intel IPP library in accordance with the specified processor type *cpu*.



WARNING. This function operates only in the static version of the library.

Return Values

<code>ippStsNoErr</code>	Indicates that the required processor-specific static code is successfully set.
<code>ippStsCpuMismatch</code>	Indicates that the specified processor type is not valid and the previously set code version is used.
<code>ippStsNoOperationInDll</code>	Indicates that there is no such operation in the dynamic version of the library.

Vector Initialization Functions



This chapter describes Intel[®] IPP functions that initialize vectors with either constants, the contents of other vectors, or the generated signals.

The full list of functions in this group is given in [Table 4-1](#).

Table 4-1 Intel IPP Vector Initialization Functions

Function Base Name	Operation
Vector Initialization Functions	
Copy	Copies the contents of one vector into another.
Move	Moves the contents of one vector to another vector.
Set	Initializes vector elements to a specified common value.
Zero	Initializes a vector to zero.
Tone and Triangle Generation Functions	
ToneInitAllocQ15	Allocates memory and initializes the tone generator state
ToneFree	Frees memory allocated by the function ippsToneInitAlloc.
ToneGetStateSizeQ15	Computes the length of the tone generator structure.
ToneInitQ15	Initializes the tone generator specification structure.
ToneQ15	Generates a tone in accordance with tone generator specification
Tone_Direct	Generates a tone with a given frequency, phase, and magnitude.
ToneQ15_Direct	Generates a tone with a given frequency, phase, and magnitude.
TriangleInitAllocQ15	Allocates memory and initializes the triangle generator state.
TriangleFree	Frees memory allocated by the function ippsTriangleInitAlloc.

Table 4-1 Intel IPP Vector Initialization Functions (continued)

Function Base Name	Operation
<u>TriangleGetStateSizeQ15</u>	Computes the length of the triangle generator structure.
<u>TriangleInitQ15</u>	Initializes the triangle generator specification structure.
<u>TriangleQ15</u>	Generates a triangle in accordance with triangle generator specification
<u>Triangle_Direct</u>	Generates a triangle with a given frequency, phase, and magnitude.
<u>TriangleQ15_Direct</u>	Generates a triangle with a given frequency, phase, and magnitude
Uniform Distribution Functions	
<u>RandUniformInitAlloc</u>	Allocates memory and initializes a noise generator with uniform distribution.
<u>RandUniformFree</u>	Closes the uniform distribution generator state.
<u>RandUniformGetSize</u>	Computes the length of the uniform distribution generator structure
<u>RandUniformInit</u>	Initializes a noise generator with uniform distribution.
<u>RandUnifrom</u>	Generates the pseudo-random samples with a uniform distribution.
<u>RandUniform_Direct</u>	Generates the pseudo-random samples with a uniform distribution in direct mode.
Gaussian Distribution Functions	
<u>RandGaussInitAlloc</u>	Allocates memory and initializes a noise generator with Gaussian distribution.
<u>RandGaussFree</u>	Closes the Gaussian distribution generator state.
<u>RandGaussGetSize</u>	Computes the length of the Gaussian distribution generator state
<u>RandGaussInit</u>	Initializes a noise generator with Gaussian distribution.
<u>RandGauss</u>	Generates the pseudo-random samples with a Gaussian distribution.
<u>RandGauss_Direct</u>	Generates pseudo-random samples with a Gaussian distribution in the direct mode.
Special Vector Functions	
<u>VectorJaehne</u>	Creates a Jaehne vector.
<u>VectorRamp</u>	Creates a ramp vector.

Vector Initialization Functions

This section describes functions that initialize the values of vector elements.

All vector elements can be initialized to a common zero or another specified value. They can also be initialized to respective values of a second vector elements.

Copy

Copies the contents of one vector into another.

Syntax

```
IppStatus ippsCopy_1u(const Ipp8u* pSrc, int srcBitOffset, Ipp8u* pDst,
    int dstBitOffset, int len);

IppStatus ippsCopy_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);

IppStatus ippsCopy_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);

IppStatus ippsCopy_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);

IppStatus ippsCopy_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);

IppStatus ippsCopy_64s(const Ipp64s* pSrc, Ipp64s* pDst, int len );

IppStatus ippsCopy_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len );

IppStatus ippsCopy_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);

IppStatus ippsCopy_32sc(const Ipp32sc* pSrc, Ipp32sc* pDst, int len);

IppStatus ippsCopy_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);

IppStatus ippsCopy_64sc(const Ipp64sc* pSrc, Ipp64sc* pDst, int len);

IppStatus ippsCopy_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

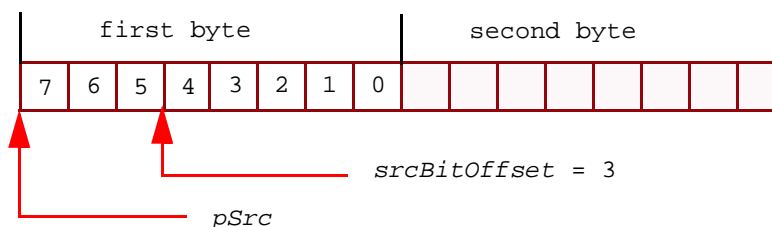
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements to copy.
<i>srcBitOffset</i>	Offset (in bits) in the first byte of the source vector.
<i>dstBitOffset</i>	Offset (in bits) in the first byte of the destination vector.

Description

The function `ippsCopy` is declared in the `ipps.h` file. This function copies the first `len` elements from a source vector `pSrc` into a destination vector `pDst`.

ippsCopy_1u. This function flavor performs copying operation on a vector that has a `8u` data type where each byte represents eight consecutive elements. In this case additional parameters `srcBitOffset` and `dstBitOffset` are required to specify start positions in the source and destination vectors respectively. Note that the bit order in each byte is inverse relative to the element order, that is, the first element in a vector is represented by the last (7th) bit in the first byte of the vector (see [Figure 4-1](#)).

Figure 4-1 Bit Layout for the Function `ippsCopy_1u`.



CAUTION. These functions perform only copying operations described above and are not intended to move data. Their behavior is unpredictable if source and destination buffers are overlapping. To move data, the functions `ippsMove` must be used

[Example 4-1](#) shows how to use the function `ippsCopy`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Example 4-1 Using the ippsCopy Function

```

IppStatus copy(void) {
    char src[] = "to be copied\0";
    char dst[256];
    return ippsCopy_8u(src, dst, strlen(src)+1);
}

```

Move

*Moves the contents of one vector
to another vector.*

Syntax

```

IppStatus ippsMove_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsMove_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMove_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsMove_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMove_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len );
IppStatus ippsMove_64s(const Ipp64s* pSrc, Ipp64s* pDst, int len);
IppStatus ippsMove_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsMove_32sc(const Ipp32sc* pSrc, Ipp32sc* pDst, int len);
IppStatus ippsMove_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsMove_64sc(const Ipp64sc* pSrc, Ipp64sc* pDst, int len);
IppStatus ippsMove_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source vector used to initialize <i>pDst</i> .
<i>pDst</i>	Pointer to the destination vector to be initialized.
<i>len</i>	Number of elements to move.

Description

The function `ippsMove` is declared in the `ipps.h` file. This function moves the first `len` elements from a source vector `pSrc` into the destination vector `pDst`. If some parts of the source and destination vectors are overlapping, then the function ensures that the original source bytes in the overlapping parts are moved (it means that they are copied before being overwritten) to the appropriate parts of the destination vector.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Set

Initializes vector elements to a specified common value.

Syntax

```

IppStatus ippsSet_8u(Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsSet_16s(Ipp16s val, Ipp16s* pDst, int len);
IppStatus ippsSet_16sc(Ipp16sc val, Ipp16sc* pDst, int len);
IppStatus ippsSet_32s(Ipp32s val, Ipp32s* pDst, int len);
IppStatus ippsSet_32f(Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsSet_32sc(Ipp32sc val, Ipp32sc* pDst, int len);
IppStatus ippsSet_32fc(Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsSet_64s(Ipp64s val, Ipp64s* pDst, int len);
IppStatus ippsSet_64f(Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsSet_64sc(Ipp64sc val, Ipp64sc* pDst, int len);
IppStatus ippsSet_64fc(Ipp64fc val, Ipp64fc* pDst, int len);

```

Parameters

<code>pDst</code>	Pointer to the vector to be initialized.
<code>len</code>	Number of elements to initialize.

val Value used to initialize the vector *pDst*.

Description

The function `ippsSet` is declared in the `ipps.h` file. This function initializes the first *len* elements of the real or complex vector *pDst* to contain the same value *val*.

[Example 4-2](#) shows how to use the function `ippsSet`.

Example 4-2 Using the `ippsSet` Function

```
IppStatus set(void) {
    char src[] = "set";
    return ippsSet_8u('0', src, strlen(src));
}
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Zero

Initializes a vector to zero.

Syntax

```
IppStatus ippsZero_8u(Ipp8u* pDst, int len);
IppStatus ippsZero_16s(Ipp16s* pDst, int len);
IppStatus ippsZero_32s(Ipp32s* pDst, int len);
IppStatus ippsZero_32f(Ipp32f* pDst, int len);
IppStatus ippsZero_64s(Ipp64s* pDst, int len);
IppStatus ippsZero_64f(Ipp64f* pDst, int len);
IppStatus ippsZero_16sc(Ipp16sc* pDst, int len);
```

```
IppStatus ippsZero_32sc(Ipp32sc* pDst, int len);
IppStatus ippsZero_32fc(Ipp32fc* pDst, int len);
IppStatus ippsZero_64sc(Ipp64sc* pDst, int len);
IppStatus ippsZero_64fc(Ipp64fc* pDst, int len);
```

Parameters

<i>pDst</i>	Pointer to the vector to be initialized to zero.
<i>len</i>	Number of elements to initialize.

Description

The function `ippsZero` is declared in the `ipps.h` file. This function initializes the first *len* elements of the vector *pDst* to 0. If *pDst* is a complex vector, both real and imaginary parts are zeroed.

[Example 4-3](#) shows how to use the function `ippsZero`.

Example 4-3 Using the `ippsZero` Function

```
IppStatus zero(void) {
    char src[] = "zero";
    return ippsZero_8u(src, strlen(src));
}
```

Return Values

<code>ippStsNoErr</code>	Indicates no error
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is <code>NULL</code>
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0

Sample-Generating Functions

This section describes Intel IPP functions which generate tone samples, triangle samples, pseudo-random samples with uniform distribution, and pseudo-random samples with Gaussian distribution, as well as special test samples.

Some sample-generating functions operate with data in the fixed point format. These functions have Q15 suffix in their name. This means that integer data are used in calculations inside the function as real numbers equal to the integer value multiplied by 2^{-15} (where “15” is called a scale factor).

Tone-Generating Functions

The functions described below generate a tone (or “sinusoid”) of a given frequency, phase, and magnitude. Tones are fundamental building blocks for analog signals. Thus, sampled tones are extremely useful in signal processing systems as test signals and as building blocks for more complex signals.

The use of tone functions is preferable against the analogous C math library’s `sin()` function for many applications, because Intel IPP functions can use information retained from the computation of the previous sample to compute the next sample much faster than standard `sin()` or `cos()`.

ToneInitAllocQ15

Allocates memory and initializes the tone generator specification structure for fixed point data.

Syntax

```
IppStatus ippSToneInitAllocQ15_16s(IppToneState_16s** pToneState, Ipp16s magn,
    Ipp16s rFreqQ15, Ipp32s phaseQ15);
```

Parameters

<i>pToneState</i>	Pointer to the pointer to the tone generator specification structure.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].

Description

The function `ippsToneInitAllocQ15` is declared in the `ipps.h` file. This function allocates memory and initializes the tone generator structure `pToneState` with the specified frequency `rFreqQ15`, phase `phaseQ15`, and magnitude `magn`.

Input data in the fixed point format Q15 are converted to the corresponding float data type that lay in the range $[0, 0.5)$ for relative frequency and $[0, 2\pi)$ for phase.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pToneState</code> pointer is <code>NULL</code> .
<code>ippStsToneMagnErr</code>	Indicates an error when <code>magn</code> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <code>rFreqQ15</code> is negative, or greater than 16383.
<code>ippStsTonePhaseErr</code>	Indicates an error when the <code>phaseQ15</code> value is negative, or greater than 205886.

ToneFree

Frees memory allocated by the function

`ippsToneInitAllocQ15`.

Syntax

```
IppStatus ippsToneFree(IppToneState_16s* pToneState);
```

Parameters

`pToneState` Pointer to the tone generator specification structure.

Description

The function `ippsToneFree` is declared in the `ipps.h` file. This function closes the tone generator state by freeing all memory associated with the structure created by [`ippsToneInitAllocQ15`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointer <i>pToneState</i> is NULL.

ToneGetStateSizeQ15

Computes the length of the tone generator structure.

Syntax

```
ippStatus ippToneGetStateSizeQ15_16s(int* pToneStateSize);
```

Parameters

<i>pToneStateSize</i>	Pointer to the computed value of size in bytes of the generator specification structure.
-----------------------	--

Description

The function `ippToneGetStateSizeQ15` is declared in the `ipps.h` file. This function computes the length (in bytes) *pToneStateSize* of the tone generator structure. The function `ippToneGetStateSizeQ15` should be called before the call to the function [ippToneInitQ15](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointer <i>pToneStateSize</i> is NULL.

ToneInitQ15

Initializes the tone generator specification structure for fixed point data.

Syntax

```
IppStatus ippsToneInitQ15_16s(IppToneState_16s* pToneState, Ipp16s magn,
    Ipp16s rFreqQ15, Ipp32s phaseQ15);
```

Parameters

<i>pToneState</i>	Pointer to the tone generator specification structure.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].

Description

The function `ippsToneInitQ15` is declared in the `ipps.h` file. This function initializes the tone generator structure *pToneState* with the specified frequency *rFreqQ15*, phase *phaseQ15*, and magnitude *magn*. The structure is allocated in the external buffer, the size of which should be computed by the function [ippsToneGetStateSizeQ15](#).

Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for relative frequency and [0, 2 π) for phase.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pToneState</i> pointer is NULL.
<code>ippStsToneMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <i>rFreqQ15</i> is negative, or greater than 16383.

`ippStsTonePhaseErr` Indicates an error when the *phaseQ15* value is negative, or greater than 205886.

ToneQ15

Generates a tone with a frequency, phase, and magnitude specified in the tone generator structure.

Syntax

```
IppStatus ippstToneQ15_16s(Ipp16s* pDst, int len, IppToneState_16s* pToneState);
```

Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.
<i>pToneState</i>	Pointer to the tone generator specification structure.

Description

The function `ippstToneQ15` is declared in the `ipps.h` file. This function generates the tone with the frequency, phase, and magnitude parameters that are specified in the previously created structure *pToneState*. The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values $x[n]$ are computed using the same [formulas](#) as in `ippstToneDirect` function for computing real tones.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pToneState</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

Tone_Direct

Generates a tone with a given frequency, phase, and magnitude.

Syntax

```
IppStatus ippsTone_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn, float rFreq,
    float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn, float
    rFreq, float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_32f(Ipp32f* pDst, int len, float magn, float rFreq,
    float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_32fc(Ipp32fc* pDst, int len, float magn, float rFreq,
    float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_64f(Ipp64f* pDst, int len, double magn, double rFreq,
    double* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_64fc(Ipp64fc* pDst, int len, double magn, double rFreq,
    double* pPhase, IppHintAlgorithm hint);
```

Parameters

<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>pPhase</i>	Pointer to the phase of the tone relative to a cosine wave. It must be in range $[0.0, 2\pi)$. The returned value may be used to compute the next continuous data block.
<i>rFreq</i>	Frequency of the tone relative to the sampling frequency. It must be in the interval $[0.0, 0.5)$ for real tone and in $[0.0, 1.0)$ for complex tone.
<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.
<i>hint</i>	Suggests using specific code. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments .”

Description

The function `ippsTone_Direct` is declared in the `ipps.h` file. This function generates the tone with the specified frequency `rFreq`, phase `pPhase`, and magnitude `magn`. The function computes `len` samples of the tone, and stores them in the array `pDst`. For real tones, each generated value `x[n]` is defined as:

$$x[n] = \text{magn} \cdot \cos(2\pi n \cdot \text{rFreq} + \text{pPhase})$$

For complex tones, `x[n]` is defined as:

$$x[n] = \text{magn} \cdot (\cos(2\pi n \cdot \text{rFreq} + \text{pPhase}) + j \cdot \sin(2\pi n \cdot \text{rFreq} + \text{pPhase}))$$

The `hint` argument suggests using specific code, which provides for either fast but less accurate calculation, or more accurate but slower execution. The values you can enter for the `hint` argument are listed in [Table 7-3](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pPhase</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.
<code>ippStsToneMagnErr</code>	Indicates an error when <code>magn</code> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <code>rFreq</code> is negative, or greater than or equal to 0.5 for real tone and to 1.0 for complex tone.
<code>ippStsTonePhaseErr</code>	Indicates an error when the <code>pPhase</code> value is negative, or greater than or equal to <code>IPP_2PI</code> .

ToneQ15_Direct

Generates a tone with a given frequency, phase, and magnitude.

Syntax

```
IppStatus ippsToneQ15_Direct_16s(Ipp16s* pDst, int len, Ipp16smagn, Ipp16srFreqQ15,
    Ipp32s phaseQ15);
```

Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].

Description

The function `ippsToneQ15_Direct` is declared in the `ipps.h` file. This function generates the tone with the specified frequency *rFreqQ15*, phase *pPhaseQ15*, and magnitude *magn*. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for relative frequency and [0, 2 π) for phase.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values *x[n]* are computed using the same [formulas](#) as in `ippsTone_Direct` function for computing real tones.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.
<code>ippStsToneMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <i>rFreqQ15</i> is negative, or greater than 16383.
<code>ippStsTonePhaseErr</code>	Indicates an error when the <i>phaseQ15</i> value is negative, or greater than 205886.

Triangle-Generating Functions

This section describes the functions that generate a periodic signal with a triangular wave form (referred to as “triangle”) of a given frequency, phase, magnitude, and asymmetry.

Application Notes

A real periodic signal with triangular wave form $x[n]$ (referred to as a real triangle) of a given frequency $rFreq$, phase value $phase$, magnitude $magn$, and asymmetry h is defined as follows:

$$x[n] = magn \cdot \mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase), \quad n = 0, 1, 2, \dots$$

A complex periodic signal with triangular wave form $x[n]$ (referred to as a complex triangle) of a given frequency $rFreq$, phase value $phase$, magnitude mag , and asymmetry h is defined as follows:

$$x[n] = magn \cdot (\mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase) + j \cdot \mathbf{st}_h(2\pi \cdot rFreq \cdot n + phase)), \quad n = 0, 1, 2, \dots$$

The $\mathbf{ct}_h()$ function is determined as follows:

$$H = \pi + h$$

$$\mathbf{ct}_h(\alpha) = \begin{cases} -\frac{2}{H} \cdot \left(\alpha - \frac{H}{2} \right), & 0 \leq \alpha \leq H \\ \frac{2}{2\pi - H} \cdot \left(\alpha - \frac{2\pi + H}{2} \right), & H \leq \alpha \leq 2\pi \end{cases}$$

$$\mathbf{ct}_h(\alpha + k \cdot 2\pi) = \mathbf{ct}_h(\alpha), \quad k = 0, \pm 1, \pm 2, \dots$$

When $H = \pi$, asymmetry $h = 0$, and function $\mathbf{ct}_h()$ is symmetric and a triangular analog of the $\cos()$ function. Note the following equations:

$$\mathbf{ct}_h(H/2 + k \cdot \pi) = 0, \quad k = 0, \pm 1, \pm 2, \dots$$

$$\mathbf{ct}_h(k \cdot 2\pi) = 1, \quad k = 0, \pm 1, \pm 2, \dots$$

$$\mathbf{ct}_h(H + k \cdot 2\pi) = -1, \quad k = 0, \pm 1, \pm 2, \dots$$

The $\mathbf{st}_h()$ function is determined as follows:

$$\mathbf{st}_h(\alpha) = \begin{cases} \frac{2}{2\pi-H} \cdot \alpha, & 0 \leq \alpha \leq \frac{2\pi-H}{2} \\ -\frac{2}{H} \cdot (\alpha - \pi), & \frac{2\pi-H}{2} \leq \alpha \leq \frac{2\pi+H}{2} \\ \frac{2}{2\pi-H} \cdot (\alpha - 2\pi), & \frac{2\pi+H}{2} \leq \alpha \leq \pi \end{cases}$$

$$\mathbf{st}_h(\alpha + k \cdot 2\pi) = \mathbf{st}_h(\alpha), \quad k = 0, \pm 1, \pm 2, \dots$$

When $H = \pi$, asymmetry $h = 0$, and function $\mathbf{st}_h(\cdot)$ is a triangular analog of a sine function. Note the following equations:

$$\mathbf{st}_h(\alpha) = \mathbf{ct}_h(\alpha + (3\pi + h)/2)$$

$$\mathbf{st}_h(\pi k) = 0, \quad k = 0, \pm 1, \pm 2, \dots$$

$$\mathbf{st}_h((\pi - h)/2 + 2\pi k) = 1, \quad k = 0, \pm 1, \pm 2, \dots$$

$$\mathbf{st}_h((3\pi + h)/2 + 2\pi k) = -1, \quad k = 0, \pm 1, \pm 2, \dots$$

TriangleInitAllocQ15

Allocates memory and initializes the triangle generator specification structure for fixed point data.

Syntax

```
IppStatus ippsTriangleInitAllocQ15_16s(IppTriangleState_16s** pTriangleState,
    Ipp16s magn, Ipp16s rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

Parameters

<i>pTriangleState</i>	Pointer to the pointer to the triangle generator specification structure.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].

asymQ15 Asymmetry h of a triangle in Q16.15 format. It must be in the range $[-102943, 102943]$. If $h=0$, then the triangle is symmetric and a direct analog of a tone.

Description

The function `ippsTriangleInitAllocQ15` is declared in the `ipps.h` file. This function allocates memory and initializes the triangle generator structure *pTriangleState* with the specified frequency *rFreqQ15*, phase *phaseQ15*, asymmetry *asymQ15* and magnitude *magn*. Input data in the fixed point format Q15 format are converted to the corresponding float data type that lay in the range $[0, 0.5)$ for the relative frequency, $[0, 2\pi)$ for the phase, and $(-\pi, \pi)$ for the asymmetry.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTriandleState</i> pointer is NULL.
<code>ippStsTriangleMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsTriangleFreqErr</code>	Indicates an error when <i>rFreqQ15</i> is negative, or greater than 16383.
<code>ippStsTrianglePhaseErr</code>	Indicates an error when the <i>phaseQ15</i> value is negative, or greater than 205886.
<code>ippStsTriangleAsymErr</code>	Indicates an error when the <i>asymQ15</i> value is less than -102943 or greater than 102943.

TriangleFree

Frees memory allocated by the function

`ippsTriangleInitAlloc`.

Syntax

```
IppStatus ippsTriangleFree(IppTriangleState_16s* pTriangleState);
```

Parameters

pTriangleState Pointer to the triangle generator specification structure.

Description

The function `ippsTriangleFree` is declared in the `ipps.h` file. This function closes the triangle generator state by freeing all memory associated with the structure created by [ippsTriangleInitAllocQ15](#).

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the pointer *pTriangleState* is NULL.

TriangleGetStateSizeQ15

Computes the length of the triangle generator structure.

Syntax

```
ippStatus ippsTriangleGetStateSizeQ15_16s(int* pTriangleStateSize);
```

Parameters

pTriangleStateSize Pointer to the computed value of size in bytes of the generator specification structure.

Description

The function `ippsTriangleGetStateSizeQ15` is declared in the `ipps.h` file. This function computes the length (in bytes) *pTriangleStateSize* of the tone generator structure. The function `ippsTriangleGetStateSizeQ15` should be called before the call to the function [ippsTriangleInitQ15](#).

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the pointer *pTriangleStateSize* is NULL.

TriangleInitQ15

Initializes the triangle generator specification structure for fixed point data.

Syntax

```
IppStatus ippsTriangleInitQ15_16s(IppTriangleState_16s* pTriangleState,
    Ipp16s magn, Ipp16s rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

Parameters

<i>pTriangleState</i>	Pointer to the triangle generator specification structure.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
<i>asymQ15</i>	Asymmetry h of a triangle in Q16.15 format. It must be in the range [-102943, 102943]. If $h=0$, then the triangle is symmetric and a direct analog of a tone.

Description

The function `ippsTriangleInitQ15` is declared in the `ipps.h` file. This function initializes the triangle generator structure *pTriangleState* with the specified magnitude *magn*, frequency *rFreqQ15*, phase *phaseQ15*, and asymmetry *asymQ15*. The structure is allocated in the external buffer, the size of which should be computed by the function [ippsTriangleGetStateSizeQ15](#). Input data in the fixed point format Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2 π) for the phase, and (- π , π) for the asymmetry.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTriangleState</i> pointer is NULL.
<code>ippStsTriangleMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsTriangleFreqErr</code>	Indicates an error when <i>rFreqQ15</i> is negative, or greater than 16383.
<code>ippStsTrianglePhaseErr</code>	Indicates an error when the <i>phaseQ15</i> value is negative, or greater than 205886.
<code>ippStsTriangleAsymErr</code>	Indicates an error when the <i>asymQ15</i> value is less than -102943 or greater than 102943.

TriangleQ15

Generates a triangle with a frequency, phase, and magnitude specified in the triangle generator structure.

Syntax

```
IppStatus ippTriangleQ15_16s(Ipp16s* pDst, int len, IppTriangleState_16s*
    pTriangleState);
```

Parameters

<i>pDst</i>	Pointer to the array which stores the generated samples.
<i>len</i>	Number of samples to be computed.
<i>pTriangleState</i>	Pointer to the triangle generator specification structure.

Description

The function `ippTriangleQ15` is declared in the `ipps.h` file. This function generates the triangle with the frequency, phase, magnitude, and asymmetry parameters that are specified in the previously created structure *pTriangleState*. The function computes *len* samples of the triangle, and stores them in the array *pDst*. Generated values $x[n]$ are computed using the same [formulas](#) as in `ippTriangle_Direct` function for computing real triangles.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pToneState</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

Triangle_Direct

Generates a triangle with a given frequency, phase, and magnitude.

Syntax

```
IppStatus ippTriangle_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn, float rFreq,
    float asym, float* pPhase);
IppStatus ippTriangle_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn, float rFreq,
    float asym, float* pPhase);
IppStatus ippTriangle_Direct_32f(Ipp32f* pDst, int len, float magn, float rFreq,
    float asym, float* pPhase);
IppStatus ippTriangle_Direct_32fc(Ipp32fc* pDst, int len, float magn, float rFreq,
    float asym, float* pPhase);
IppStatus ippTriangle_Direct_64f(Ipp64f* pDst, int len, double magn, double rFreq,
    double asym, double* pPhase);
IppStatus ippTriangle_Direct_64fc(Ipp64fc* pDst, int len, double magn, double rFreq,
    double asym, double* pPhase);
```

Parameters

<i>rFreq</i>	Frequency of the triangle relative to the sampling frequency. It must be in range [0.0, 0.5).
<i>pPhase</i>	Pointer to the phase of the triangle relative to a cosine triangular analog wave. It must be in range [0.0, 2π). The returned value may be used to compute the next continuous data block.
<i>magn</i>	Magnitude of the triangle, that is, the maximum value attained by the wave.

<i>asym</i>	Asymmetry h of a triangle. It must be in range $[-\pi, \pi)$. If $h=0$, then the triangle is symmetric and a direct analog of a tone.
<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.

Description

The function `ippsTriangle` is declared in the `ipps.h` file. This function generates the triangle with the specified frequency *rFreq*, phase pointed by *pPhase*, and magnitude *magn*. The function computes *len* samples of the triangle, and stores them in the array *pDst*. For real triangle, $x[n]$ is defined as:

$$x[n] = \text{magn} \cdot \text{ct}_h(2\pi \cdot \text{rFreq} \cdot n + \text{phase}), \quad n = 0, 1, 2, \dots$$

For complex triangles, $x[n]$ is defined as:

$$x[n] = \text{magn} \cdot (\text{ct}_h(2\pi \cdot \text{rFreq} \cdot n + \text{phase}) + j \cdot \text{st}_h(2\pi \cdot \text{rFreq} \cdot n + \text{phase})), \quad n = 0, 1, 2, \dots$$

See “Application Notes” on [page 4-17](#) for the definition of functions `cth` and `sth`.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pPhase</i> pointer is NULL.
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.
<code>ippsStsTrnglMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippsStsTrnglFreqErr</code>	Indicates an error when <i>rFreq</i> is negative, or greater than or equal to 0.5.
<code>ippsStsTrnglPhaseErr</code>	Indicates an error when the <i>pPhase</i> value is negative, or greater than or equal to <code>IPP_2PI</code> .
<code>ippsStsTrnglAsymErr</code>	Indicates an error when <i>asym</i> is less than <code>-IPP_PI</code> , or greater than or equal to <code>IPP_PI</code> .

TriangleQ15_Direct

Generates a triangle with a given frequency, phase, and magnitude for fixed point data.

Syntax

```
IppStatus ippsTriangleQ15_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn, Ipp16s
    rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
<i>asymQ15</i>	Asymmetry h of a triangle in Q16.15 format. It must be in the range [-102943, 102943]. If $h=0$, then the triangle is symmetric and a direct analog of a tone.

Description

The function `ippsTriangleQ15_Direct` is declared in the `ipps.h` file. This function generates the triangle with the specified magnitude *magn*, frequency *rFreqQ15*, phase *pPhaseQ15*, and asymmetry *asymQ15*. Input data in the fixed point format Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2π) for the phase, and $[-\pi, \pi)$ for the asymmetry.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values $x[n]$ are computed using the same [formulas](#) as in `ippsTriangle_Direct` function for computing real triangles.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.
<code>ippStsToneMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <i>rFreqQ15</i> is negative, or greater than 16383.
<code>ippStsTonePhaseErr</code>	Indicates an error when the <i>phaseQ15</i> value is negative, or greater than 205886.
<code>ippStsTriangleAsymErr</code>	Indicates an error when the <i>asymQ15</i> value is less than -102943 or greater than 102943.

Uniform Distribution Functions

This section describes the functions that generate pseudo-random samples with uniform distribution.

RandUniformInitAlloc

Allocates memory and initializes a noise generator with uniform distribution.

Syntax

```
IppStatus ippRandUniformInitAlloc_8u(IppsRandUniState_8u** pRandUniState,
    Ipp8u low, Ipp8u high, unsigned int seed);

IppStatus ippRandUniformInitAlloc_16s(IppsRandUniState_16s** pRandUniState,
    Ipp16s low, Ipp16s high, unsigned int seed);

IppStatus ippRandUniformInitAlloc_32f(IppsRandUniState_32f** pRandUniState,
    Ipp32f low, Ipp32f high, unsigned int seed);
```

Parameters

<i>pRandUniState</i>	Pointer to the structure containing parameters for the generator of noise.
----------------------	--

<i>low</i>	Lower bound of the uniform distribution range.
<i>high</i>	Upper bound of the uniform distribution range.
<i>seed</i>	Seed value used by the pseudo-random number generation algorithm.

Description

The function `ippsRandUniformInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the pseudo-random generator state `pRandUniState`. The uniform distribution range is specified by the lower and upper bounds *low* and *high*, respectively.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRandUniState</code> pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for the operation.

RandUniformFree

Closes the uniform distribution generator state.

Syntax

```
IppStatus ippsRandUniformFree_8u(IppsRandUniState_8u* pRandUniState);
IppStatus ippsRandUniformFree_16s(IppsRandUniState_16s* pRandUniState);
IppStatus ippsRandUniformFree_32f(IppsRandUniState_32f* pRandUniState);
```

Parameters

<i>pRandUniState</i>	Pointer to the structure containing parameters for the generator of noise.
----------------------	--

Description

The function `ippsRandUniformFree` is declared in the `ipps.h` file. This function closes the noise generator state `pRandUniState` by freeing all memory allocated by the `ippsRandUniInitAlloc` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pRandUniState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

RandUniformInit

Initializes a noise generator with uniform distribution.

Syntax

```
IppStatus ippRandUniformInit_16s(IppsRandUniState_16s* pRandUniState, Ipp16s low,
    Ipp16s high, unsigned int seed);
```

Parameters

<i>pRandUniState</i>	Pointer to the structure containing parameters for the generator of noise.
<i>low</i>	Lower bound of the uniform distribution range.
<i>high</i>	Upper bound of the uniform distribution range.
<i>seed</i>	Seed value used by the pseudo-random number generation algorithm.

Description

The function `ippRandUniformInit` is declared in the `ipps.h` file. This function initializes the pseudo-random generator state *pRandUniState* in the external buffer. The size of this buffer should be computed previously by calling the function [ippsRandUniformGetSize](#). The uniform distribution range is specified by the lower and upper bounds *low* and *high*, respectively.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pRandUniState</i> pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for the operation.

RandUniformGetSize

Computes the length of the uniform distribution generator structure.

Syntax

```
IppStatus ippsRandUniformGetSize_16s(int* pRandUniStateSize);
```

Parameters

pRandUniStateSize Pointer to the computed value of size in bytes of the generator specification structure.

Description

The function `ippsRandUniformGetSize` is declared in the `ipps.h` file. This function computes the length (in bytes) *pRandUniStateSize* of the uniform distribution generator structure. The function `ippsRandUniformGetSize` should be called prior to the function [`ippsRandUniformInit`](#).

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the pointer *pRandUniStateSize* is NULL.

RandUnifrom

Generates the pseudo-random samples with a uniform distribution.

Syntax

```
IppStatus ippsRandUniform_8u(Ipp8u* pDst, int len, IppsRandUniState_8u*  
    pRandUniState);  
  
IppStatus ippsRandUniform_16s(Ipp16s* pDst, int len, IppsRandUniState_16s*  
    pRandUniState);
```



```
IppStatus ippsRandUniform_32f(Ipp32f* pDst, int len, IppsRandUniState_32f*
    pRandUniState);
```

Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.
<i>pRandUniState</i>	Pointer to the structure containing parameters for the generator of noise.

Description

The function `ippsRandUniform` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a uniform distribution and stores them in the array *pDst*. Initial parameters of the generator are set in the generator state structure *pRandUniState*. Before calling `ippsRandUniform`, you must initialize the generator state by calling the [ippsRandUniformInitAlloc](#) function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pRandUniState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

RandUniform_Direct

Generates the pseudo-random samples with a uniform distribution in direct mode.

Syntax

```
IppStatus ippsRandUniform_Direct_16s(Ipp16s* pDst, int len, Ipp16s low, Ipp16s high,
    unsigned int* pSeed);
IppStatus ippsRandUniform_Direct_32f(Ipp32f* pDst, int len, Ipp32f low, Ipp32f high,
    unsigned int* pSeed);
IppStatus ippsRandUniform_Direct_64f(Ipp64f* pDst, int len, Ipp64f low, Ipp64f high,
    unsigned int* pSeed);
```

Parameters

<i>pSeed</i>	Pointer to the seed value used by the pseudo-random number generation algorithm.
<i>low</i>	Lower bound of the uniform distribution range.
<i>high</i>	Upper bound of the uniform distribution range.
<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.

Description

The function `ippsRandUniform_Direct` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a uniform distribution and stores them in the array *pDst*. This function does not require to initialize the generator state structure in advance. All parameters of the pseudo-random number generator are set directly in the function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSeed</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Gaussian Distribution Functions

This section describes the function that generates pseudo-random samples with Gaussian distribution.

RandGaussInitAlloc

Allocates memory and initializes a noise generator with Gaussian distribution.

```
IppStatus ippsRandGaussInitAlloc_8u(IppsRandGaussState_8u** pRandGaussState, Ipp8u
    mean, Ipp8u stdDev, unsigned int seed);
IppStatus ippsRandGaussInitAlloc_16s(IppsRandGaussState_16s** pRandGaussState,
    Ipp16s mean, Ipp16s stdDev, unsigned int seed);
```

```
IppStatus ippsRandGaussInitAlloc_32f(IppsRandGaussState_32f** pRandGaussState,
    Ipp32f mean, Ipp32f stdDev, unsigned int seed);
```

Parameters

<i>pRandGaussState</i>	Pointer to the structure containing parameters for the generator of noise.
<i>mean</i>	Mean of the Gaussian distribution.
<i>stdDev</i>	Standard deviation of the Gaussian distribution.
<i>seed</i>	Seed value used by the pseudo-random number generator algorithm.

Description

The function `ippsRandGaussInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the pseudo-random generator state structure *pRandGaussState*. This structure contains parameters of the required noise generator that are specified by the *mean*, *stdDev* and *seed* values.

Return Values

<code>ppStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pRandGaussState</i> pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for the operation.

RandGaussFree

Closes the Gaussian distribution generator state.

Syntax

```
IppStatus ippsRandGaussFree_8u(IppsRandGaussState_8u* pRandGaussState);
IppStatus ippsRandGaussFree_16s(IppsRandGaussState_16s* pRandGaussState);
IppStatus ippsRandGaussFree_32f(IppsRandGaussState_32f* pRandGaussState);
```

Parameters

<i>pRandGaussState</i>	Pointer to the structure containing parameters for the generator of noise.
------------------------	--

Description

The function `ippsRandGaussFree` is declared in the `ipps.h` file. This function closes the noise generator state `pRandGaussState` by freeing all memory allocated by the `ippsRandGaussInitAlloc` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRandGaussState</code> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

RandGaussGetSize

Computes the length of the Gaussian distribution generator structure.

Syntax

```
ippStatus ippsRandGaussGetSize_16s(int* pRandGaussStateSize);
```

Parameters

<code>pRandGaussStateSize</code>	Pointer to the computed value of size in bytes of the generator specification structure.
----------------------------------	--

Description

The function `ippsRandGaussGetSize` is declared in the `ipps.h` file. This function computes the length (in bytes) `pRandGaussStateSize` of the uniform distribution generator structure. The function `ippsRandGaussGetSize` should be called before calling the function [`ippsRandGaussInit`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointer <code>pRandGaussStateSize</code> is NULL.

RandGaussInit

Initializes a noise generator with Gaussian distribution.

Syntax

```
IppStatus ippsRandGaussInit_16s(IppsRandGaussState_16s* pRandGaussState, Ipp16s  
    mean, Ipp16s stdDev, unsigned int seed);
```

Parameters

<i>pRandGaussState</i>	Pointer to the structure containing parameters for the generator of noise.
<i>mean</i>	Mean of the Gaussian distribution.
<i>stdDev</i>	Standard deviation of the Gaussian distribution.
<i>seed</i>	Seed value used by the pseudo-random number generator algorithm.

Description

The function `ippsRandGaussInit` is declared in the `ipps.h` file. This function initializes the pseudo-random generator state structure *pRandGaussState* in the external buffer. The size of this buffer should be computed previously by calling the function [ippsRandGaussGetSize](#). This structure contains parameters of the required noise generator that are specified by the *mean*, *stdDev* and *seed* values.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pRandGaussState</i> pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for the operation.

RandGauss

Generates the pseudo-random samples with a Gaussian distribution.

Syntax

```
IppStatus ippsRandGauss_8u(Ipp8u* pDst, int len, IppsRandGaussState_8u*
    pRandGaussState);
IppStatus ippsRandGauss_16s(Ipp16s* pDst, int len, IppsRandGaussState_16s*
    pRandGaussState);
IppStatus ippsRandGauss_32f(Ipp32f* pDst, int len, IppsRandGaussState_32f*
    pRandGaussState);
```

Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.
<i>pRandGaussState</i>	Pointer to the structure containing parameters of the noise generator.

Description

The function `ippsRandGauss` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a Gaussian distribution and stores them in the array *pDst*. The initial parameters of the generator are set in the generator state structure *pRandGaussState*. Before calling `ippsRandGauss`, you must initialize the generator state by calling the [ippsRandGaussInitAlloc](#) function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pRandGaussState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

RandGauss_Direct

Generates pseudo-random samples with a Gaussian distribution in the direct mode.

Syntax

```
IppStatus ippsRandGauss_Direct_16s(Ipp16s* pDst, int len, Ipp16s mean,
    Ipp16s stdev, unsigned int* pSeed);
IppStatus ippsRandGauss_Direct_32f(Ipp32f* pDst, int len, Ipp32f mean,
    Ipp32f stdev, unsigned int* pSeed);
IppStatus ippsRandGauss_Direct_64f(Ipp64f* pDst, int len, Ipp64f mean,
    Ipp64f stdev, unsigned int* pSeed);
```

Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>pSeed</i>	Pointer to the seed value used by the pseudo-random number generation algorithm.
<i>len</i>	Number of samples to be computed.
<i>mean</i>	Mean of the Gaussian distribution.
<i>stdev</i>	Standard deviation of the Gaussian distribution.

Description

The function `ippsRandGauss` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a Gaussian distribution, and stores them in the array *pDst*. This function does not require to initialize the generator state structure in advance. All parameters of the pseudo-random number generator are set directly in the function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSeed</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Special Vector Functions

The functions described in this section create special vectors that can be used as a test signals to examine the effect of applying different signal processing functions.

VectorJaehne

Creates a Jaehne vector.

Syntax

```
IppStatus ippsVectorJaehne_8u(Ipp8u* pDst, int len, Ipp8u magn);
IppStatus ippsVectorJaehne_8s(Ipp8s* pDst, int len, Ipp8s magn);
IppStatus ippsVectorJaehne_16u(Ipp16u* pDst, int len, Ipp16u magn);
IppStatus ippsVectorJaehne_16s(Ipp16s* pDst, int len, Ipp16s magn);
IppStatus ippsVectorJaehne_32u(Ipp32u* pDst, int len, Ipp32u magn);
IppStatus ippsVectorJaehne_32s(Ipp32s* pDst, int len, Ipp32s magn);
IppStatus ippsVectorJaehne_32f(Ipp32f* pDst, int len, Ipp32f magn);
IppStatus ippsVectorJaehne_64f(Ipp64f* pDst, int len, Ipp64f magn);
```

Parameters

<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector.
<i>magn</i>	Magnitude of the signal to be generated.

Description

The function `ippsVectorJaehne` is declared in the `ipps.h` file. This function creates a Jaehne vector and stores the result in *pDst*. The magnitude *magn* must be positive. The function generates the sinusoid with a variable frequency. The computation is performed as follows:

$$pDst[n] = magn \cdot \sin\left(\frac{0.5\pi n^2}{len}\right), \quad 0 \leq n < len$$

[Example 4-4](#) shows how to use the function `ippsVectorJaehne`.

Example 4-4 Using the `ippsVectorJaehne` Function

```
IppStatus Jaehne (void) {
    Ipp16s buf[100] ;
    return ippsVectorJaehne_16s ( buf, 100, 255 );
}
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsJaehneErr</code>	Indicates an error when <i>magn</i> is negative.

VectorRamp

Creates a ramp vector.

Syntax

```
IppStatus ippsVectorRamp_8u(Ipp8u* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_8s(Ipp8s* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_16u(Ipp16u* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_16s(Ipp16s* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_32u(Ipp32u* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_32s(Ipp32s* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_32f(Ipp32f* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_64f(Ipp64f* pDst, int len, float offset, float slope);
```

Parameters

<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector.
<i>offset</i>	Offset value.

slope Slope coefficient.

Description

The function `ippsVectorRamp` is declared in the `ipps.h` file. This function creates a ramp vector and stores the result in *pDst*. The destination vector elements are computed according to the following formula:

$$pDst[n] = offset + slope * n, \quad 0 \leq n < len.$$

Note that linear transform coefficients *offset* and *slope* have floating-point values for all function flavors.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Essential Vector Functions

5

This chapter describes Intel[®] IPP functions that perform logical and shift, arithmetic, conversion, windowing, and statistical operations.

The full list of functions in this group is given in [Table 5-1](#).

Table 5-1 Intel IPP Essential Vector Functions

Function Base Name	Operation
Logical and Shift Functions	
AndC	Computes the bitwise AND of a scalar value and each element of a vector.
And	Computes the bitwise AND of two vectors.
OrC	Computes the bitwise OR of a scalar value and each element of a vector.
Or	Computes the bitwise OR of two vectors.
XorC	Computes the bitwise XOR of a scalar value and each element of a vector.
Xor	Computes the bitwise XOR of two vectors.
Not	Computes the bitwise NOT of the vector elements.
LShiftC	Shifts bits in vector elements to the left.
RShiftC	Shifts bits in vector elements to the right.
Arithmetic Functions	
AddC	Adds a constant value to each element of a vector.
Add	Adds the elements of two vectors.
AddProduct	Adds product of two vectors to the accumulator vector.

Table 5-1 Intel IPP Essential Vector Functions (continued)

Function Base Name	Operation
<u>AddProductC</u>	Adds product of a vector and a constant to the accumulator vector.
<u>MulC</u>	Multiplies each elements of a vector by a constant value.
<u>Mul</u>	Multiplies the elements of two vectors.
<u>SubC</u>	Subtracts a constant value from each element of a vector.
<u>SubCRev</u>	Subtracts each element of a vector from a constant value.
<u>Sub</u>	Subtracts the elements of two vectors.
<u>DivC</u>	Divides each element of a vector by a constant value.
<u>DivCRev</u>	Divides a constant value by each element of a vector.
<u>Div</u>	Divides the elements of two vectors.
<u>Abs</u>	Computes absolute values of vector elements.
<u>Sqr</u>	Computes a square of each element of a vector.
<u>Sqrt</u>	Computes a square root of each element of a vector.
<u>Cubrt</u>	Computes cube root of each element of a vector.
<u>Exp</u>	Computes e to the power of each element of a vector.
<u>Ln</u>	Computes the natural logarithm of each element of a vector.
<u>10Log10</u>	Computes the decimal logarithm of each element of a vector and multiplies it by 10.
<u>SumLn</u>	Sums natural logarithms of each element of a vector.
<u>Arctan</u>	Computes the inverse tangent of each element of a vector.
<u>Normalize</u>	Normalizes elements of a real or complex vector using offset and division operations.
Conversion Functions	
<u>SortAscend,</u> <u>SortDescend</u>	Sorts all elements of a vector.
<u>SwapBytes</u>	Reverses byte order of a vector.
<u>Convert</u>	Converts the data type of a vector and stores the results in a second vector.
<u>Join</u>	Converts the floating-point data of several vectors to integer data, and stores the results in a single vector.
<u>JoinScaled</u>	Converts with scaling the floating-point data of several vectors to integer data, and stores the results in a single vector.

Table 5-1 Intel IPP Essential Vector Functions (continued)

Function Base Name	Operation
<u>SplitScaled</u>	Converts with scaling the integer data of a vector to floating-point data, and stores the result in several vectors.
<u>Conj</u>	Stores the complex conjugate values of a vector in a second vector or in-place.
<u>ConjFlip</u>	Computes the complex conjugate of a vector and stores the result in reverse order.
<u>Magnitude</u>	Computes the magnitudes of the elements of a complex vector.
<u>MagSquared</u>	Computes the squared magnitudes of the elements of a complex vector.
<u>Phase</u>	Computes the phase angles of elements of a complex vector.
<u>PowerSpectr</u>	Computes the power spectrum of a complex vector.
<u>Real</u>	Returns the real part of a complex vector in a second vector.
<u>Imag</u>	Returns the imaginary part of a complex vector in a second vector.
<u>RealToCplx</u>	Returns a complex vector constructed from the real and imaginary parts of two real vectors.
<u>CplxToReal</u>	Returns the real and imaginary parts of a complex vector in two respective vectors.
<u>Threshold</u> <u>Threshold_LT,</u> <u>Threshold_GT</u> <u>Threshold_LTVal,</u> <u>Threshold_GTVal,</u> <u>Threshold_LTValGTVal</u>	Performs the threshold operation on the elements of a vector by limiting the element values by <i>level</i> .
<u>Threshold_LTAbs,</u> <u>Threshold_GTAbs</u>	Performs the threshold operation on the absolute values of elements of a vector.
<u>Threshold_LTInv</u>	Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.
<u>CartToPolar</u>	Converts the elements of a complex vector to polar coordinate form.
<u>PolarToCart</u>	Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.
<u>MaxOrder</u>	Computes the maximum order of a vector.
<u>Preemphasize</u>	Computes preemphasis of a single precision real signal in-place.

Table 5-1 Intel IPP Essential Vector Functions (continued)

Function Base Name	Operation
Flip	Reverses the order of elements in a vector.
FindNearestOne	Finds an element of the table which is closest to the specified value.
FindNearest	Finds table elements that are closest to the elements of the specified vector.
Viterbi Decoder Functions	
GetVarPointDV	Fills the array with the information about points that are closest to the received point.
CalcStatesDV	Calculates the states of the Viterbi decoder.
BuildSymb1TableDV4D	Fills the array with the information about possible 4D symbols.
UpdatePathMetricsDV	Searches for the state with the minimum path metric.
Windowing Functions	
WinBartlett	Multiplies a vector by a Bartlett windowing function.
WinBlackman	Multiplies a vector by a Blackman windowing function.
WinHamming	Multiplies a vector by a Hamming windowing function.
WinHann	Multiplies a vector by a Hann windowing function.
WinKaiser	Multiplies a vector by a Kaiser windowing function.
Statistical Functions	
Sum	Computes the sum of the elements of a vector.
Max	Returns the maximum value of a vector.
MaxIndx	Returns the maximum value of a vector and the index of the maximum element.
MaxAbs	Returns the maximum absolute value of a vector.
MaxAbsIndx	Returns the maximum absolute value of a vector and the index of the corresponding element
Min	Returns the minimum value of a vector.
MinIndx	Returns the minimum value of a vector and the index of the minimum element.
MinAbs	Returns the minimum absolute value of a vector.
MinAbsIndx	Returns the minimum absolute value of a vector and the index of the corresponding element

Table 5-1 Intel IPP Essential Vector Functions (continued)

Function Base Name	Operation
MinMax	Returns the maximum and minimum values of a vector.
MinMaxIndx	Returns the maximum and minimum values of a vector and the indexes of the corresponding elements.
Mean	Computes the mean value of a vector.
StdDev	Computes the standard deviation value of a vector.
Norm	Computes the C, L1, L2, or L2Sqr norm of a vector.
NormDiff	Computes the C, L1, L2, or L2Sqr norm of two vectors' difference.
DotProd	Computes the dot product of two vectors.
MaxEvery, MinEvery	Computes maximum or minimum value for each pair of elements of two vectors.
Sampling Functions	
SampleUp	Up-samples a signal, conceptually increasing its sampling rate by an integer factor.
SampleDown	Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.

Logical and Shift Functions

This section describes the Intel IPP signal processing functions that perform logical and shift operations on vectors. Logical and shift functions are only defined for integer arguments.

For binary logical operations AND, OR and XOR, the following functions are provided:

AndC, OrC, XorC for vector-scalar operations

And, Or, Xor for vector-vector operations.

AndC

Computes the bitwise AND of a scalar value and each element of a vector.

Syntax

```
IppStatus ippsAndC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsAndC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsAndC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int len);
IppStatus ippsAndC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsAndC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsAndC_32u_I(Ipp32u val, Ipp32u* pSrcDst, int len);
```

Parameters

<i>val</i>	Input scalar value.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsAndC` is declared in the `ipps.h` file. This function computes the bitwise AND of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsAndC` compute the bitwise AND of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

And

Computes the bitwise AND of two vectors.

Syntax

```

IppStatus ippsAnd_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len);
IppStatus ippsAnd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsAnd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst, int len);
IppStatus ippsAnd_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsAnd_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsAnd_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);

```

Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the two source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector for the in-place operation.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsAnd` is declared in the `ipps.h` file. This function computes the bitwise AND of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsAnd` compute the bitwise AND of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

OrC

Computes the bitwise OR of a scalar value and each element of a vector.

Syntax

```
IppStatus ippsOrC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsOrC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsOrC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int len);
IppStatus ippsOrC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsOrC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsOrC_32u_I(Ipp16u val, Ipp32u* pSrcDst, int len);
```

Parameters

<i>val</i>	Input scalar value.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsOrC` is declared in the `ipps.h` file. This function computes the bitwise OR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsOrC` compute the bitwise OR of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Or

Computes the bitwise OR of two vectors.

Syntax

```

IppStatus ippsOr_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len);
IppStatus ippsOr_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsOr_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst, int len);
IppStatus ippsOr_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsOr_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsOr_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);

```

Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the two source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector for the in-place operation.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsOr` is declared in the `ipps.h` file. This function computes the bitwise OR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsOr` compute the bitwise OR of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

XorC

Computes the bitwise XOR of a scalar value and each element of a vector.

Syntax

```
IppStatus ippsXorC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsXorC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsXorC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int len);
IppStatus ippsXorC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsXorC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsXorC_32u_I(Ipp32u val, Ipp32u* pSrcDst, int len);
```

Parameters

<i>val</i>	Input scalar value.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsXorC` is declared in the `ipps.h` file. This function computes the bitwise XOR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsXorC` compute the bitwise XOR of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Xor

Computes the bitwise XOR of two vectors.

Syntax

```
IppStatus ippsXor_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len);
IppStatus ippsXor_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsXor_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst, int len);
IppStatus ippsXor_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsXor_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsXor_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the two source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector for the in-place operation.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsXor` is declared in the `ipps.h` file. This function computes the bitwise XOR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsXor` compute the bitwise XOR of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Not

Computes the bitwise NOT of the vector elements.

Syntax

```
IppStatus ippNot_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippNot_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippNot_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippNot_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippNot_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippNot_32u_I(Ipp32u* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippNot` is declared in the `ipp.h` file. This function computes the bitwise NOT of the corresponding elements of the vectors *pSrc*, and stores the result in the vector *pDst*.

The in-place flavors of `ippNot` compute the bitwise NOT of the corresponding elements of the vector *pSrcDst* and store the result in the vector *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

LShiftC

Shifts bits in vector elements to the left.

Syntax

```

IppStatus ippsLShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int len);
IppStatus ippsLShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst, int len);
IppStatus ippsLShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst, int len);
IppStatus ippsLShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst, int len);
IppStatus ippsLShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
IppStatus ippsLShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippsLShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippsLShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);

```

Parameters

<i>val</i>	Number of bits by which the function shifts each element of the vector <i>pSrc</i> or <i>pSrcDst</i> .
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsLShiftC` is declared in the `ipps.h` file. This function shifts each element of the vector *pSrc* by *val* bits to the left, and stores the result in *pDst*.

The in-place flavors of `ippsLShiftC` shift each element of the vector *pSrcDst* by *val* bits to the left and store the result in *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.

`ippStsSizeErr` Indicates an error when *len* is less than or equal to 0.

RShiftC

Shifts bits in vector elements to the right.

Syntax

```
IppStatus ippSRShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int len);
IppStatus ippSRShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst, int len);
IppStatus ippSRShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst, int len);
IppStatus ippSRShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst, int len);
IppStatus ippSRShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
IppStatus ippSRShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippSRShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippSRShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);
```

Parameters

<i>val</i>	Number of bits by which the function shifts each element of the vector <i>pSrc</i> or <i>pSrcDst</i> .
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippSRShiftC` is declared in the `ipps.h` file. This function shifts each element of the vector *pSrc* by *val* bits to the right, and stores the result in *pDst*.

The in-place flavors of `ippSRShiftC` shift each element of the vector *pSrcDst* by *val* bits to the right and store the result in *pSrcDst*.

Note that the arithmetic shift is realized for signed data, and the logical shift for unsigned data.

[Example 5-1](#) shows how the logical and shift functions can be used in the saturate operation. The data are converted to the unsigned char range [0...255].

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Example 5-1 Using the Logical and Shift Functions

```
void saturate(void) {
    Ipp16s x[8] = {1000, -257, 127, 4, 5, 0, 7, 8}, lo[8], hi[8];
    IppStatus status = ippNot_16u((Ipp16u*)x, (Ipp16u*)lo, 8);
    ippRShiftC_16s_I(15, lo, 8);
    ippCopy_16s(x, hi, 8);
    ippSubCRev_16s_ISfs(255, hi, 8, 0);
    ippRShiftC_16s_I(15, hi, 8);
    ippAnd_16u_I((Ipp16u*)lo, (Ipp16u*)x, 8);
    ippOr_16u_I((Ipp16u*)hi, (Ipp16u*)x, 8);
    ippAndC_16u_I(255, (Ipp16u*)x, 8);
    printf_16s("saturate =", x, 8, status);
}
```

Output:

```
saturate = 255 0 127 4 5 0 7 8
```

Arithmetic Functions

This section describes the Intel IPP signal processing functions that perform vector arithmetic operations on vectors. The arithmetic functions include basic element-wise arithmetic operations between vectors, as well as more complex calculations such as computing absolute values, square and square root, natural logarithm and exponential of vector elements.

Intel IPP software provides two versions of each function. One version performs the operation in-place, while the other stores the results of the operation in a different destination vector, that is, executes an out-of-place operation.

AddC

Adds a constant value to each element of a vector.

Syntax

```

IppStatus ippsAddC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsAddC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsAddC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsAddC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
IppStatus ippsAddC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsAddC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsAddC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsAddC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsAddC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsAddC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsAddC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsAddC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len,
    int scaleFactor);
IppStatus ippsAddC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len,
    int scaleFactor);
IppStatus ippsAddC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst,
    int len, int scaleFactor);
IppStatus ippsAddC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst,
    int len, int scaleFactor);
IppStatus ippsAddC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAddC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int
    scaleFactor);

```

Parameters

<i>val</i>	Scalar value used to increment each element of the vector <i>pSrc</i> or <i>pSrcDst</i> .
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector which stores the result of the addition $pSrc[n] + val$.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsAddC` is declared in the `ipps.h` file. This function adds a value *val* to each element of the source vector *pSrc*, and stores the result in the destination vector *pDst*.

The in-place flavors of `ippsAddC` add a value *val* to each element of the vector *pSrcDst*, and store the result in *pSrcDst*.

Functions with *Sfs* suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Add

Adds the elements of two vectors.

Syntax

```
IppStatus ippsAdd_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst, int len);
IppStatus ippsAdd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len);
```

```

IppStatus ippsAdd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst, int len);
IppStatus ippsAdd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, int len);
IppStatus ippsAdd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, int len);
IppStatus ippsAdd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
    int len);
IppStatus ippsAdd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
    int len);
IppStatus ippsAdd_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsAdd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f* pDst,
    int len);
IppStatus ippsAdd_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsAdd_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsAdd_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsAdd_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsAdd_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsAdd_16s32s_I(const Ipp16s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsAdd_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsAdd_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, Ipp32sc*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsAdd_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAdd_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAdd_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    scaleFactor);

```

```
IppStatus ippsAdd_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAdd_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len, int
    scaleFactor)
```

Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the vectors whose elements are to be added together.
<i>pDst</i>	Pointer to the destination vector which stores the result of the addition $pSrc2[n] + pSrc1[n]$.
<i>pSrc</i>	Pointer to the source vector whose elements are to be added to the elements of <i>pSrcDst</i> in-place.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsAdd` is declared in the `ipps.h` file. This function adds the elements of the vector *pSrc1* to the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors of `ippsAdd` add the elements of the vector *pSrc* to the elements of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

[Example 5-2](#) shows that overflow does not occur while adding big numbers due to the scaling operation.

Return Values

<code>ippStsNoErr</code>	Indicates no error
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code>
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0

Example 5-2 Using the ippsAdd Function

```
IppStatus add(void) {
    Ipp16s x[4] = {-1, 32767, 2, 30000};
    IppStatus st = ippsAdd_16s_ISfs(x, x, 4, 1);
    printf_16s("add =", x, 4, st);
    return st;
}
```

Output:
add = -1 32767 2 30000

AddProduct

Adds product of two vectors to the accumulator vector.

Syntax

```
IppStatus ippsAddProduct_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pSrcDst,
    int len);
IppStatus ippsAddProduct_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pSrcDst,
    int len);
IppStatus ippsAddProduct_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
    pSrcDst, int len);
IppStatus ippsAddProduct_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
    pSrcDst, int len);
IppStatus ippsAddProduct_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pSrcDst, int len, int scaleFactor);
IppStatus ippsAddProduct_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
    pSrcDst, int len, int scaleFactor);
IppStatus ippsAddProduct_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32s*
    pSrcDst, int len, int scaleFactor);
```

Parameters

pSrc1, pSrc2

Pointers to the source vectors.

<i>pSrcDst</i>	Pointer to the destination accumulator vector.
<i>len</i>	The number of elements in the vectors.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsAddProduct` is declared in the `ipps.h` file. This function multiplies each element of the source vector *pSrc1* by the corresponding element of the vector *pSrc2*, and adds the result to the corresponding element of the accumulator vector *pSrcDst* as given by:

$$pSrcDst[n] = pSrcDst[n] + pSrc1[n] * pSrc2[n], \quad 0 \leq n < len$$

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

AddProductC

Adds product of a vector and a constant to the accumulator vector.

Syntax

```
IppStatus ippsAddProductC_32f(const Ipp32f* pSrc, const Ipp32f val,
                              Ipp32f* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointers to the source vector.
<i>val</i>	The scalar value used to multiply each element of the source vector <i>pSrc</i> .
<i>pSrcDst</i>	Pointer to the destination accumulator vector.
<i>len</i>	The number of elements in the vectors.

The function `ippsAddProductC` is declared in the `ipps.h` file. This function multiplies each element of the source vector `pSrc` by a value `val` and adds the result to the corresponding element of the accumulator vector `pSrcDst` as given by:

$$pSrcDst[n] = pSrcDst[n] + pSrc[n] * val, \quad 0 \leq n < len$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if any of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>len</code> is less than or equal to 0.

MulC

Multiplies each elements of a vector by a constant value.

Syntax

```

IppStatus ippsMulC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsMulC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsMulC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsMulC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
IppStatus ippsMulC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsMulC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsMulC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsMulC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsMulC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsMulC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int
    scaleFactor);
IppStatus ippsMulC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsMulC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len,
    int scaleFactor);
IppStatus ippsMulC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len,
    int scaleFactor);

```



```

IppStatus ippsMulC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst, int
    len, int scaleFactor);
IppStatus ippsMulC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst, int
    len, int scaleFactor);
IppStatus ippsMulC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_16sc_ISfs(Ipp16scval, Ipp16sc* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_32sc_ISfs(Ipp32scval, Ipp32sc* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_32f16s_Sfs(const Ipp32f* pSrc, Ipp32f val, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsMulC_Low_32f16s(const Ipp32f* pSrc, Ipp32f val, Ipp16s* pDst,
    int len);

```

Parameters

<i>val</i>	The scalar value used to multiply each element of the vector <i>pSrc</i> or <i>pSrcDst</i> .
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector which stores the result of the multiplication <i>pSrc[n] * val</i> .
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	The number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMulC` is declared in the `ipps.h` file. This function multiplies each element of the vector *pSrc* by a value *val* and stores the result in *pDst*.

The in-place flavors of `ippsMulC` multiply each element of the vector *pSrcDst* by a value *val* and store the result in *pSrcDst*.

Function flavor with `Low` suffix in its name requires that each value of the product *pSrc*val* does not exceed the `Ipp32s` data type range.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the `scaleFactor` value. If the output value exceeds the data range, the result becomes saturated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Mul

Multiplies the elements of two vectors.

Syntax

```

IppStatus ippsMul_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst,
    int len);
IppStatus ippsMul_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
    int len);
IppStatus ippsMul_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
    int len);
IppStatus ippsMul_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
    pDst, int len);
IppStatus ippsMul_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
    pDst, int len);
IppStatus ippsMul_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp16u* pDst,
    int len);
IppStatus ippsMul_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f* pDst,
    int len);
IppStatus ippsMul_32f32fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
    int len);

IppStatus ippsMul_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMul_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMul_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsMul_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);

```

```

IppStatus ippsMul_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsMul_32f32fc_I(const Ipp32f* pSrc, Ipp32fc* pSrcDst, int len);

IppStatus ippsMul_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int
    len, int scaleFactor);
IppStatus ippsMul_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst,
    int len, int scaleFactor);
IppStatus ippsMul_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsMul_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s* pDst,
    int len, int scaleFactor);
IppStatus ippsMul_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, Ipp32sc*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_16u16s_Sfs(const Ipp16u* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsMul_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32s* pDst,
    int len, int scaleFactor);
IppStatus ippsMul_32s32sc_Sfs(const Ipp32s* pSrc1, const Ipp32sc* pSrc2, Ipp32sc*
    pDst, int len, int scaleFactor);

IppStatus ippsMul_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsMul_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_32s32sc_ISfs(const Ipp32s* pSrc, Ipp32sc* pSrcDst, int len, int
    scaleFactor);

```

```
IppStatus ippsMul_Low_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
    pDst, int len, int scaleFactor);
```

Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the vectors whose elements are to be multiplied.
<i>pDst</i>	Pointer to the destination vector which stores the result of the multiplication <i>pSrc1</i> [<i>n</i>] * <i>pSrc2</i> [<i>n</i>].
<i>pSrc</i>	Pointer to the vector whose elements are to be multiplied by the elements of <i>pSrcDst</i> in-place.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMul` is declared in the `ipps.h` file. This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2* and stores the result in *pDst*.

The in-place flavors of `ippsMul` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Function flavor with `Low` suffix in its name requires that each value of the product does not exceed the `Ipp32s` data type range.

Return Values

<code>ippStsNoErr</code>	Indicates no error
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code>
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0

SubC

Subtracts a constant value from each element of a vector.

Syntax

```
IppStatus ippsSubC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsSubC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsSubC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsSubC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
IppStatus ippsSubC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsSubC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsSubC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int
    scaleFactor);
IppStatus ippsSubC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len,
    int scaleFactor);
IppStatus ippsSubC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsSubC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len,
    int scaleFactor);
IppStatus ippsSubC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst, int
    len, int scaleFactor);
IppStatus ippsSubC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst, int
    len, int scaleFactor);
IppStatus ippsSubC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int scaleFactor);
```

Parameters

<i>val</i>	Scalar value used to decrement each element of the vector <i>pSrc</i> or <i>pSrcDst</i> .
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector which stores the result of the subtraction $pSrc[n] - val$.
<i>pSrcDst</i>	Pointer to the vector whose elements are to be decreased by the value <i>val</i> and stored in <i>pSrcDst</i> for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSubC` is declared in the `ipps.h` file. This function subtracts a value *val* from each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsSubC` subtract a value *val* from each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

SubCRev

Subtracts each element of a vector from a constant value.

Syntax

```
IppStatus ippsSubCRev_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
```

```

IppStatus ippsSubCRev_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsSubCRev_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsSubCRev_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
IppStatus ippsSubCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubCRev_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubCRev_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubCRev_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsSubCRev_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsSubCRev_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubCRev_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int
    len, int scaleFactor);
IppStatus ippsSubCRev_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int
    len, int scaleFactor);
IppStatus ippsSubCRev_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int
    len, int scaleFactor);
IppStatus ippsSubCRev_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst,
    int len, int scaleFactor);
IppStatus ippsSubCRev_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst,
    int len, int scaleFactor);
IppStatus ippsSubCRev_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubCRev_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubCRev_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubCRev_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSubCRev_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubCRev_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int
    scaleFactor);

```

Parameters

<i>val</i>	Scalar value from which vector elements are subtracted.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.

<i>pSrcDst</i>	Pointer to the vector whose elements are to be subtracted from the value <i>val</i> in case of the in-place operation. The destination vector which stores the result of the subtraction $val - pSrcDst[n]$.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSubCRev` is declared in the `ipps.h` file. This function subtracts each element of the vector *pSrc* from a value *val* and stores the result in *pDst*.

The in-place flavors of `ippsSubCRev` subtract each element of the vector *pSrcDst* from a value *val* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Sub

Subtracts the elements of two vectors.

Syntax

```
IppStatus ippsSub_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst, int len);
IppStatus ippsSub_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, int len);
IppStatus ippsSub_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, int len);
IppStatus ippsSub_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
    int len);
IppStatus ippsSub_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
    int len);
```



```

IppStatus ippsSub_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f* pDst,
    int len);
IppStatus ippsSub_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsSub_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsSub_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsSub_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsSub_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsSub_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsSub_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u*
    pDst, int len, int scaleFactor);
IppStatus ippsSub_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsSub_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
    pDst, int len, int scaleFactor);
IppStatus ippsSub_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc*
    pDst, int len, int scaleFactor);
IppStatus ippsSub_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, Ipp32sc*
    pDst, int len, int scaleFactor);
IppStatus ippsSub_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSub_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSub_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSub_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSub_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSub_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len, int
    scaleFactor);

```

Parameters

<i>pSrc1</i>	Pointer to the source vector whose elements are to be subtracted from <i>pSrc2</i> .
<i>pSrc2</i>	Pointer to the source vector whose elements are to be decreased by the elements of <i>pSrc1</i> .

<i>pDst</i>	Pointer to the destination vector which stores the result of the subtraction $pSrc2[n] - pSrc1[n]$.
<i>pSrc</i>	Pointer to the vector whose elements are to be subtracted from the elements of <i>pSrcDst</i> in-place.
<i>pSrcDst</i>	Pointer to the vector whose elements are to be decreased by the elements of <i>pSrc</i> and stored in <i>pSrcDst</i> for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSub` is declared in the `ipps.h` file. This function subtracts the elements of the vector *pSrc1* from the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors of `ippsSub` subtract the elements of the vector *pSrc* from the elements of a vector *pSrcDst* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

<code>ippStsNoErr</code>	Indicates no error
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code>
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0

DivC

Divides each element of a vector by a constant value.

Syntax

```
IppStatus ippsDivC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsDivC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsDivC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsDivC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
IppStatus ippsDivC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
```

```

IppStatus ippsDivC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsDivC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsDivC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsDivC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len,
    int ScaleFactor);
IppStatus ippsDivC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int
    len, int ScaleFactor);
IppStatus ippsDivC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst,
    int len, int ScaleFactor);
IppStatus ippsDivC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int ScaleFactor);
IppStatus ippsDivC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int ScaleFactor);
IppStatus ippsDivC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int ScaleFactor);

```

Parameters

<i>val</i>	Scalar value used to divide each element of the vector <i>pSrc</i> or <i>pSrcDst</i> .
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector which stores the result of the division <i>pSrc[n]/val</i> .
<i>pSrcDst</i>	Pointer to the vector whose elements are divided by the value <i>val</i> and stored in <i>pSrcDst</i> for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDivC` is declared in the `ipps.h` file. This function divides each element of the vector *pSrc* by a value *val* and stores the result in *pDst*.

The in-place flavors of `ippsDivC` divide each element of the vector *pSrcDst* by a value *val* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDivByZeroErr</code>	Indicates an error when <i>val</i> is equal to 0.

DivCRev

Divides a constant value by each element of a vector.

Syntax

```

IppStatus ippsDivCRev_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsDivCRev_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsDivCRev_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsDivCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);

```

Parameters

<i>val</i>	Constant value that is used as a dividend in the operation.
<i>pSrc</i>	Pointer to the source vector whose elements are used as divisors.
<i>pDst</i>	Pointer to the destination vector which stores the result of the division $val / pSrc[n]$.
<i>pSrcDst</i>	Pointer to the vector whose elements are used both as divisors for the in-place operation and to store the resulting quotients.
<i>len</i>	Number of elements in the vector

Description

The function `ippsDivCRev` is declared in the `ipps.h` file. This function divides the constant value *val* by each element of the vector *pSrc* and stores the results in *pDst*.

The in-place flavors of `ippsDivC` divide the constant value *val* by each element of the vector *pSrcDst* and store the results in *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDivByZeroErr</code>	Indicates an error when <i>val</i> is equal to 0.

Div

Divides the elements of two vectors.

Syntax

```

IppStatus ippDiv_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, int len);
IppStatus ippDiv_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, int len);
IppStatus ippDiv_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
    int len);
IppStatus ippDiv_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
    int len);
IppStatus ippDiv_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippDiv_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippDiv_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippDiv_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippDiv_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int
    len, int scaleFactor);
IppStatus ippDiv_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippDiv_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s* pDst,
    int len, int scaleFactor);
IppStatus ippDiv_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc*
    pDst, int len, int scaleFactor);
IppStatus ippDiv_32s16s_Sfs(const Ipp16s* pSrc1, const Ipp32s* pSrc2, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippDiv_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int ScaleFactor);

```

```
IppStatus ippsDiv_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    ScaleFactor);
IppStatus ippsDiv_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
    ScaleFactor);
IppStatus ippsDiv_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    ScaleFactor);
```

Parameters

<i>pSrc1</i>	Pointer to the vector whose elements are used as divisors for the elements of <i>pSrc2</i> .
<i>pSrc2</i>	Pointer to the vector whose elements are to be divided by the elements of <i>pSrc1</i> .
<i>pDst</i>	Pointer to the destination vector <i>pDst</i> which stores the result of the division $pSrc2[n] / pSrc1[n]$.
<i>pSrc</i>	Pointer to the source vector whose elements are used as divisors for the elements of <i>pSrcDst</i> in case of the in-place operation.
<i>pSrcDst</i>	Pointer to the vector whose elements are to be divided by the elements of <i>pSrc</i> and stored in <i>pSrcDst</i> for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDiv` is declared in the `ipps.h` file. This function divides the elements of the vector *pSrc2* by the elements of the vector *pSrc1*, and stores the result in *pDst*.

The in-place flavors of `ippsDiv` divide the elements of the vector *pSrcDst* by the elements of the vector *pSrc* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

If the function `ippsDiv` encounters a zero-valued divisor vector element, it returns a warning status and continues execution with the corresponding result value (see appendix A [“Handling of Special Cases”](#) for more information).

[Example 5-3](#) shows that the usage of the scaling factor in the integer functions increases operation accuracy. [Example 5-4](#) considers division by zero exceptions ($x / 0, 0 / 0$).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning for zero-valued divisor vector element.

Example 5-3 Using the `ippsDiv_16s_ISfs` Function

```
IppStatus div16s( void ) {  
    Ipp16s x[4] = { -3, 2, 0, 300 };  
    Ipp16s y[4] = { -2, 2, 0, 0 };  
    IppStatus st = ippsDiv_16s_ISfs( y, x, 4, -1 );  
    printf_16s("div16s =", x, 4, st );  
    return st;  
}
```

Output:

```
-- warning 6, Zero value(s) in the divisor of the function Div  
div16s =  3 2 0 32767
```

Example 5-4 Using the ippsDiv_32f_I Function

```
IppStatus div32f( void ) {
    Ipp32f x[4] = { -3, 2, 0, 300 };
    Ipp32f y[4] = { -2, 2, 0, 0 };
    IppStatus st = ippsDiv_32f_I( y, x, 4 );
    printf_32f( "div32f =", x, 4, st );
    return st;
}
```

Output:

```
-- warning 6, Zero value(s) in the divisor of the function Div
div32f =  1.500000 1.000000 1.#IND00 1.#INF00
```

Abs

Computes absolute values of vector elements.

Syntax

```
IppStatus ippsAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAbs_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsAbs_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsAbs_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsAbs_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.

len Number of elements in the vector.

Description

The function `ippsAbs` is declared in the `ipps.h` file. This function computes the absolute values of each element of the vector *pSrc* and stores the result in *pDst*.

The in-place flavors of `ippsAbs` compute the absolute values of each element of the vector *pSrcDst* and store the result in *pSrcDst*.

To compute the absolute values of complex data, use the function `ippsMagnitude`.

[Example 5-5](#) shows how to call the function `ippsAbs_32f_I`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Example 5-5 Using the `ippsAbs` Function

```
void abs32f(void) {
    Ipp32f x[4] = {-1, 1, 0, 0};
    x[3] *= (-1);
    ippsAbs_32f_I(x, 4);
    printf_32f("abs =", x, 4, ippStsNoErr);
}

Output:
abs =  1.000000 1.000000 0.000000 0.000000
```

Sqr

Computes a square of each element of a vector.

Syntax

```
IppStatus ippsSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```

IppStatus ippsSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqr_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqr_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqr_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqr_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqr_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqr_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16sc_ISfs(Ipp16sc* pSrcDst, int len, int scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSqr` is declared in the `ipps.h` file. This function computes the square of each element of the vector *pSrc*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = pSrc[n]^2$$

The in-place flavors of `ippsSqr` compute the square of each element of the vector `pSrcDst` and store the result in `pSrcDst`. The computation is performed as follows:

$$pSrcDst[n] = pSrcDst[n]^2$$

When computing the square of an integer number, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. [Example 5-6](#) shows how to get the value of 200^2 . Without scaling this result is clipped to 32767. Scaling retains the output data range but results in the precision loss in low-order bits.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Example 5-6 Using the `ippsSqr` Function

```
IppStatus sqr(void) {
    Ipp16s x[4] = {-3, 2, 30, 200};
    IppStatus st = ippsSqr_16s_ISfs(x, 4, 1);
    printf_16s("sqr =", x, 4, st);
    return st;
}
```

Output:

```
sqr =  4 2 450 20000
```

Sqrt

Computes a square root of each element of a vector.

Syntax

```
IppStatus ippsSqrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqrt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqrt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
```

```

IppStatus ippsSqrt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqrt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqrt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqrt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqrt_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqrt_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len, int scaleFactor);
IppStatus ippsSqrt_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSqrt_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len, int scaleFactor);
IppStatus ippsSqrt_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, int
    scaleFactor);
IppStatus ippsSqrt_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len, int scaleFactor);
IppStatus ippsSqrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsSqrt_64s16s_Sfs(const Ipp64s* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsSqrt_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16sc_ISfs(Ipp16sc* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSqrt` is declared in the `ipps.h` file. This function computes the square root of each element of the vector *pSrc*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = \sqrt{pSrc[n]}$$

The in-place flavors of `ippsSqrt` compute the square root of each element of the vector `pSrcDst` and store the result in `pSrcDst`. The computation is performed as follows:

$$pSrcDst[n] = \sqrt{pSrcDst[n]}$$

The square root of complex vector elements is computed as follows:

$$\sqrt{a + j \cdot b} = \sqrt{\frac{\sqrt{a^2 + b^2} + a}{2}} + j \cdot \text{sign}(b) \cdot \sqrt{\frac{\sqrt{a^2 + b^2} - a}{2}}$$

Application Notes

If the function `ippsSqrt` encounters a negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A [“Handling of Special Cases”](#) for more information).

To increase precision of an integer output, use the scale factor.

[Example 5-7](#) shows how to call the function `ippsSqrt_16s_ISfs`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

`ippStsSqrtNegArg` Indicates a warning that a source element has a negative value.

Example 5-7 Using the `ippsSqrt` Function

```
IppStatus sqrt(void) {
    Ipp16s x[4] = {-3, 2, 30, 300};
    IppStatus st = ippsSqrt_16s_ISfs(x, 4, -1);
    printf_16s("sqrt =", x, 4, st);
    return st;
}
```

Output:

```
-- warning 3, Negative value(s) in the argument of the function Sqrt
sqrt = 0 3 11 35
```

Cubrt

Computes cube root of each element of a vector.

Syntax

```
IppStatus ippsCubrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCubrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsCubrt` is declared in the `ipps.h` file. This function computes cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

$$pDst[n] = \sqrt[3]{pSrc[n]}, 0 \leq n < len.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pSrc</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Exp

Computes e to the power of each element of a vector.

Syntax

```

IppStatus ippExp_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippExp_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippExp_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippExp_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippExp_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippExp_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippExp_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippExp_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len, int scaleFactor);
IppStatus ippExp_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippExp_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippExp_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);

```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>pSrcDst</code>	Pointer to the source and destination vector <code>pSrcDst</code> for the in-place operation.
<code>len</code>	Number of elements in the vector

scaleFactor Refer to [Integer Scaling](#).

Description

The function `ippsExp` is declared in the `ipps.h` file. This function computes the exponential function of each element of the vector *pSrc*, and stores the result in *pDst*.

The computation is performed as follows:

$$pDst[n] = e^{pSrc[n]}$$

The in-place flavors of `ippsExp` compute the exponential function of each element of the vector *pSrcDst* and store the result in *pSrcDst*.

The computation is performed as follows:

$$pSrcDst[n] = e^{pSrcDst[n]}$$

When an overflow occurs, the function continues operation with the corresponding result value (see appendix A [“Handling of Special Cases”](#) for more information).

When computing the exponent of an integer number, the output result can exceed the data range and become saturated. The scaling retains the output data range but results in precision loss in low-order bits. The function `ippsExp_32f64f` computes the output result in a higher precision data range.

[Example 5-8](#) shows how to call the function `ippsExp_16s_ISfs`. [Example 5-9](#) shows how to call the function `ippsExp_64f_I`.

Application Notes

For the functions `ippsExp` and `ippsLn` the result is rounded to the nearest integer after scaling.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Example 5-8 Using the `ippsExp_16s_ISfs` Function

```
IppStatus expl6s(void) {  
    Ipp16s x[4] = {-1, 2, 30, 0};  
    IppStatus st = ippsExp_16s_ISfs(x, 4, -1);  
    printf_16s("expl6s =", x, 4, st);  
    return st;  
}
```

Output:

```
expl6s =  1 15 32767 2
```

Example 5-9 Using the `ippsExp_64f_I` Function

```
IppStatus exp64f(void) {  
    Ipp64f x[4] = {-1, 2, 1, log(1.234567)};  
    IppStatus st = ippsExp_64f_I(x, 4);  
    printf_64f("exp64f =", x, 4, st);  
    return st;  
}
```

Output:

```
exp64f =  0.367879 7.389056 2.718282 1.234567
```

Ln

Computes the natural logarithm of each element of a vector.

Syntax

```
IppStatus ippsLn_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsLn_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsLn_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsLn_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsLn_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsLn_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsLn_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsLn` is declared in the `ipps.h` file. This function computes the natural logarithm of each element of the vector *pSrc* and stores the result in *pDst* as given by

$$pDst[n] = \log_e(pSrc[n])$$

The in-place flavors of `ippsLn` compute the natural logarithm of each element of the vector *pSrcDst* and store the result in *pSrcDst* as given by

$$pSrcDst[n] = \log_e(pSrcDst[n])$$

If the function `ippsLn` encounters a zero or negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A [“Handling of Special Cases”](#) for more information).

[Example 5-10](#) shows how to call the function `ippsLn_32f_I`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements.
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements.

Example 5-10 Using the `ippsLn` Function

```
IppStatus ln32f(void) {
    Ipp32f x[4] = {-1, (float)IPP_E, 0, (float)(exp(1.234567))};
    IppStatus st = ippsLn_32f_I(x, 4);
    printf_32f("Ln = ", x, 4, st);
    return st;
}
```

Output:

```
-- warning 8, Negative value(s) of argument in the Ln function
Ln = -1.#IND00 1.000000 -1.#INF00 1.234567
```

10Log10

Computes the decimal logarithm of each element of a vector and multiplies it by 10.

Syntax

```
IppStatus ipps10Log10_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int
    scaleFactor);
IppStatus ipps10Log10_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ipps10Log10` is declared in the `ipps.h` file. This function computes the decimal logarithm of each element of the vector *pSrc*, multiplies it by 10, and stores the result in *pDst* as given by

$$pDst[n] = 10 * \log_{10} (pSrc[n])$$

The in-place flavor of `ipps10Log10` computes the decimal logarithm of each element of the vector *pSrcDst*, multiplies it by 10, and stores the result in *pSrcDst* as given by

$$pSrcDst[n] = 10 * \log_{10} (pSrcDst[n])$$

If the function `ipps10Log10` encounters a zero or negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A [“Handling of Special Cases”](#) for more information).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements.
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements.

SumLn

Sums natural logarithms of each element of a vector.

Syntax

```

IppStatus ippsSumLn_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum);
IppStatus ippsSumLn_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_32f64f(const Ipp32f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pSum);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSum</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsSumLn` is declared in the `ipps.h` file. This function computes the sum of natural logarithms of each element of the vector *pSrc* and stores the result value in *pSum*. The summation is given by:

$$sum = \sum_{n=0}^{len-1} \ln(pSrc[n])$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pSum</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>-Inf</code> .
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>NaN</code> .

Arctan

Computes the inverse tangent of each element of a vector.

Syntax

```

IppStatus ippArctan_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippArctan_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippArctan_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippArctan_64f_I(Ipp64f* pSrcDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector <i>pSrcDst</i> for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippArctan` is declared in the `ipps.h` file. This function computes the inverse tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

$$pDst[n] = \arctan(pSrc[n]), \quad 0 \leq n < len.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Normalize

Normalizes elements of a real or complex vector using offset and division operations.

Syntax

```

IppStatus ippNormalize_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f vsub,
    Ipp32f vdiv);
IppStatus ippNormalize_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f vsub,
    Ipp64f vdiv);
IppStatus ippNormalize_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, Ipp32fc
    vsub, Ipp32f vdiv);
IppStatus ippNormalize_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, Ipp64fc
    vsub, Ipp64f vdiv);
IppStatus ippNormalize_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
    vsub, int vdiv, int scaleFactor);
IppStatus ippNormalize_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, Ipp16sc
    vsub, int vdiv, int scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>vsub</i>	Subtrahend value.
<i>vdiv</i>	Denominator value.
<i>pDst</i>	Pointer to the vector which stores the normalized elements.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsNormalize` is declared in the `ipps.h` file. This function subtracts `vsub` from elements of the input vector `pSrc`, divides the differences by `vdiv`, and stores the result in `pDst`. The computation is performed as follows:

$$pDst[n] = \frac{pSrc[n] - vsub}{vdiv}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsDivByZeroErr</code>	Indicates an error when <code>vdiv</code> is equal to 0 or less than the minimum floating-point positive number.

Conversion Functions

The functions described in this section perform the following conversion operations for vectors:

- Sorting all elements of a vector
- Data type conversion (including floating-point to integer and integer to floating-point)
- Joining several vectors
- Extracting components from a complex vector and constructing a complex vector
- Computing the complex conjugates of vectors
- Cartesian to polar and polar to Cartesian coordinate conversion.

This section also describes the Intel IPP functions that extract real and imaginary components from a complex vector or construct a complex vector using its real and imaginary components. The functions `ippsReal` and `ippsImag` return the real and imaginary parts of a complex vector in a separate vector, respectively.

The function `ippsRealToCplx` constructs a complex vector from real and imaginary components stored in two respective vectors.

The function `ippsCplxToReal` returns the real and imaginary parts of a complex vector in two respective vectors.

The function `ippsMagnitude` computes the magnitude of a complex vector elements.

Additionally this section describes functions that perform the Viterbi decoding for V34 receiver.

SortAscend, SortDescend

Sorts all elements of a vector.

Syntax

```
IppStatus ippsSortAscend_8u_I(Ipp8u* pSrcDst, int len);  
IppStatus ippsSortAscend_16s_I(Ipp16s* pSrcDst, int len);  
IppStatus ippsSortAscend_32s_I(Ipp32s* pSrcDst, int len);  
IppStatus ippsSortAscend_32f_I(Ipp32f* pSrcDst, int len);  
IppStatus ippsSortAscend_64f_I(Ipp64f* pSrcDst, int len);  
  
IppStatus ippsSortDescend_8u_I(Ipp8u* pSrcDst, int len);  
IppStatus ippsSortDescend_16s_I(Ipp16s* pSrcDst, int len);  
IppStatus ippsSortDescend_32s_I(Ipp32s* pSrcDst, int len);  
IppStatus ippsSortDescend_32f_I(Ipp32f* pSrcDst, int len);  
IppStatus ippsSortDescend_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector.
<i>len</i>	Number of elements in the vector.

Description

The functions `ippsSortAscend` and `ippsSortDescend` are declared in the `ipps.h` file. These functions rearrange all elements of the specified vector *pSrcDst* in the ascending or descending order, respectively, and store the result in the destination vector *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

SwapBytes

Reverses the byte order of a vector.

Syntax

```

IppStatus ippSwapBytes_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippSwapBytes_24u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippSwapBytes_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippSwapBytes_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippSwapBytes_24u_I(Ipp8u* pSrcDst, int len);
IppStatus ippSwapBytes_32u_I(Ipp32u* pSrcDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippSwapBytes` is declared in the `ipps.h` file. This function reverses the endian order (byte order) of the source vector *pSrc* (*pSrcDst* for the in-place operation) and stores the result in *pDst* (*pSrcDst*).

When the low-order byte is stored in memory at the lowest address, and the high-order byte at the highest address, the little-endian order is implemented. When the high-order byte is stored in memory at the lowest address, and the low-order byte at the highest address, the big-endian order is implemented. The function `ippSwapBytes` allows to switch from one order to the other in either direction.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Convert

Converts the data type of a vector and stores the results in a second vector.

Syntax

```
IppStatus ippsConvert_8s16s(const Ipp8s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsConvert_8s32f(const Ipp8s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsConvert_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16u32f(const Ipp16u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s16s(const Ipp32s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsConvert_32s32f(const Ipp32s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s64f(const Ipp32s* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32f_Sfs(const Ipp16s* pSrc, Ipp32f* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_16s64f_Sfs(const Ipp16s* pSrc, Ipp64f* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32s32f_Sfs(const Ipp32s* pSrc, Ipp32f* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32s64f_Sfs(const Ipp32s* pSrc, Ipp64f* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32f8s_Sfs(const Ipp32f* pSrc, Ipp8s* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f8u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16s_Sfs(const Ipp32f* pSrc, Ipp16s* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16u_Sfs(const Ipp32f* pSrc, Ipp16u* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
```

```

IppStatus ippsConvert_32f32s_Sfs(const Ipp32f* pSrc, Ipp32s* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_64s32s_Sfs(const Ipp64s* pSrc, Ipp32s* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_64f32s_Sfs(const Ipp64f* pSrc, Ipp32s* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);

IppStatus ippsConvert_24u32u(const Ipp8u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsConvert_24u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32u24u_Sfs(const Ipp32u* pSrc, Ipp8u* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32f24u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len, int
    scaleFactor);

IppStatus ippsConvert_24s32s(const Ipp8u* pSrc, Ipp32s* pDst, int len);
IppStatus ippsConvert_24s32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s24s_Sfs(const Ipp32s* pSrc, Ipp8u* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32f24s_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len, int
    scaleFactor);

IppStatus ippsConvert_16s16f(const Ipp16s* pSrc, Ipp16f* pDst, int len,
    IppRoundMode rndMode);
IppStatus ippsConvert_32f16f(const Ipp32f* pSrc, Ipp16f* pDst, int len,
    IppRoundMode rndMode);
IppStatus ippsConvert_16f16s_Sfs(const Ipp16f* pSrc, Ipp16s* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_16f32f(const Ipp16f* pSrc, Ipp32f* pDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>rndMode</i>	Rounding mode which can be <code>ippRndZero</code> or <code>ippRndNear</code> :
<code>ippRndZero</code>	Specifies that floating-point values must be truncated toward zero.

	<code>ippRndNear</code>	Specifies that floating-point values must be rounded to the nearest integer.
<code>len</code>	Number of elements in the vector	
<code>scaleFactor</code>	Refer to Integer Scaling in Chapter 2.	

Description

The function `ippsConvert` is declared in the `ipps.h` file. This function converts the type of data contained in the vector `pSrc` and stores the results in `pDst`.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the `scaleFactor` value. The converted result is saturated if it exceeds the output data range.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Join

Converts the floating-point data of several vectors to integer data, and stores the results in a single vector.

Syntax

```
IppStatus ippsJoin_32f16s_D2L(const Ipp32f** pSrc, int nChannels, int chanLen, Ipp16s* pDst);
```

Parameters

<code>pSrc</code>	Pointer to an array of pointers to the input vectors.
<code>pDst</code>	Pointer to the output vector.
<code>nChannels</code>	Number of input vectors.
<code>chanLen</code>	Number of elements in each input vector.

Description

The function `ippsJoin` is declared in the `ipps.h` file. This function converts floating-point data of `nChannels` input vectors stored in the array `pSrc` to integer data type and writes the results to the destination vector `pDst` in the following order: first element of first vector, first element of second vector, ..., first element of the last vector in the array; second element of first vector, second element of second vector, and so on.

The converted value is saturated if it exceeds the output data range.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error condition if <code>nChannels</code> or <code>chanLen</code> is less than or equal to 0.

JoinScaled

Converts with scaling the floating-point data of several vectors to integer data and stores the results in a single vector.

Syntax

```
IppStatus ippsJoinScaled_32f16s_D2L(const Ipp32f** pSrc, int nChannels, int chanLen,
    Ipp16s* pDst);
```

```
IppStatus ippsJoinScaled_32f24s_D2L(const Ipp32f** pSrc, int nChannels, int chanLen,
    Ipp8u* pDst);
```

Parameters

<code>pSrc</code>	Pointer to an array of pointers to the input vectors.
<code>pDst</code>	Pointer to the output vector.
<code>nChannels</code>	Number of input vectors.
<code>chanLen</code>	Number of elements in each input vector.

Description

The function `ippsJoinScaled` is declared in the `ipps.h` file. This function converts floating-point data of `nChannels` input vectors stored in the array `pSrc` to integer data type with scaling. Input data must be in the range `[-1.0, 1.0]` that is mapped onto output data type range `[dst_Min..dst_Max]` (see [Data Ranges](#) in Chapter 2 for more information on data types and ranges). If the input data exceed the range `[-1.0, 1.0]`, they are saturated.

The function uses the following formula for scaling:

$$pDst[n] = dst_Min + k * [pSrc[n] - (-1)]$$

$$\text{where } k = (dst_Max - dst_Min) / (1 - (-1))$$

Mapped values are rounded to the nearest integer.

The function `ippsJoinScaled` writes the results to the destination vector `pDst` in the following order: first element of first vector, first element of second vector, ..., first element of the last vector in the array; second element of first vector, second element of second vector, and so on.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates a then error condition if at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error condition if <code>nChannels</code> or <code>chanLen</code> is less than or equal to 0.

SplitScaled

Converts the integer data of a vector to floating-point data with scaling, and stores the result in several vectors.

Syntax

```
IppStatusippsSplitScaled_16s32f_D2L(constIpp16s*pSrc,Ipp32f**pDst,intnChannels,
int chanLen);

IppStatusippsSplitScaled_24s32f_D2L(constIpp8u*pSrc,Ipp32f**pDst,intnChannels,
int chanLen);
```

Parameters

<i>pSrc</i>	Pointer to the input vector.
<i>pDst</i>	Pointer to an array of pointers to the output vectors.
<i>nChannels</i>	Number of output vectors.
<i>chanLen</i>	Number of elements in each output vector.

Description

The function `ippsSplitScaled` is declared in the `ipps.h` file. This function converts integer data of the input vector *pSrc* to the floating-point data type with scaling. The whole range of the input data [*src_Min*..*src_Max*] (see [Data Ranges](#) in Chapter 2 for more information on data types and ranges) is mapped onto the output data range [-1.0, 1.0]

The function uses the following formula for scaling:

$$pDst[n] = (-1) + k * (pSrc[n] - src_Min)$$

$$\text{where } k = (1 - (-1)) / (src_Max - src_Min)$$

The function `ippsSplitScaled` writes the results to the *nChannels* destination vectors *pDst* in the following order: first element of the input vector becomes the first element of the first output vector, second element of the input vector becomes the first element of the second output vector, and so on.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates a then error condition if at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error condition if <i>nChannels</i> or <i>chanLen</i> is less than or equal to 0.

Conj

Stores the complex conjugate values of a vector in a second vector or in-place.

Syntax

```
IppStatus ippsConj_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsConj_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsConj_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsConj_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsConj_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsConj_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsConj` is declared in the `ipps.h` file. This function stores in *pDst* the element-wise conjugation of the complex vector *pSrc*. The element-wise conjugation of the vector is defined as follows:

$$pDst[n].re = pSrc[n].re$$

$$pDst[n].im = - pSrc[n].im$$

The in-place flavors of `ippsConj` store in *pSrcDst* the element-wise conjugation of the complex vector *pSrcDst*.

The element-wise conjugation of the vector is defined as follows:

$$pSrcDst[n].re = pSrcDst[n].re$$

$$pSrcDst[n].im = - pSrcDst[n].im$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

ConjFlip

Computes the complex conjugate of a vector and stores the result in reverse order.

Syntax

```

IppStatus ippConjFlip_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippConjFlip_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippConjFlip_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector.

Description

The function `ippConjFlip` is declared in the `ipps.h` file. This function computes the conjugate of the vector *pSrc* and stores the result, in reverse order, in *pDst*.

The complex conjugate, stored in reverse order, is defined as follows:

$$pDst[n] = conj(pSrc[len - n - 1])$$

Note that if *pSrc* and *pDst* overlap in memory, the function returns unpredictable results.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.

`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0.

Magnitude

Computes the magnitudes of the elements of a complex vector.

Syntax

```
IppStatus ippMagnitude_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
    pDst, int len);
IppStatus ippMagnitude_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
    pDst, int len);
IppStatus ippMagnitude_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippMagnitude_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippMagnitude_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp32f* pDst, int len);
IppStatus ippMagnitude_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippMagnitude_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippMagnitude_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippMagnitude_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len,
    int scaleFactor);
```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pSrcRe</code>	Pointer to the vector with the real parts of complex elements.
<code>pSrcIm</code>	Pointer to the vector with the imaginary parts of complex elements.
<code>pDst</code>	Pointer to the destination vector.
<code>len</code>	Number of elements in the vector
<code>scaleFactor</code>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMagnitude` is declared in the `ipps.h` file. The complex flavor of this function computes the element-wise magnitude of the complex vector `pSrc` and stores the result in `pDst`. The element-wise magnitude is defined by the formula:

$$\text{magn}[n] = \sqrt{pSrc[n].re^2 + pSrc[n].im^2}$$

The real flavor of the function `ippsMagnitude` computes the element-wise magnitude of the complex vector whose real and imaginary components are specified in the vectors `pSrcRe` and `pSrcIm`, respectively, and stores the result in `pDst`. The element-wise magnitude is defined by the formula:

$$\text{magn}[n] = \sqrt{pSrcRe[n]^2 + pSrcIm[n]^2}$$

[Example 5-11](#) verifies the identity $\sin^2 x + \cos^2 x = 1$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Example 5-11 Using the `ippsMagnitude` Function

```
void magn(void) {
    Ipp64f x[6], magn[4];
    int n;
    for (n = 0; n<6; ++n) x[n] = sin(IPP_2PI * n / 8);
    ippsMagnitude_64f(x, x+2, magn, 4);
    printf_64f("magn =", magn, 4, ippStsNoErr);
}
```

Output:

```
magn = 1.000000 1.000000 1.000000 1.000000
```

Matlab* Analog:

```
>> n = 0:9; x = sin(2*pi*n/8); z = [x(1:8)+j*x(3:10)]; abs(z(1:4))
```

MagSquared

Computes the squared magnitudes of the elements of a complex vector.

Syntax

```
IppStatus ippsMagSquared_32sc32s_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len,  
int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMagSquared` is declared in the `ipps.h` file. This function computes the element-wise squared magnitude of the complex vector *pSrc* and stores the result in *pDst*. The element-wise squared magnitude is defined by the formula:

$$magn[n] = pSrc[n].re^2 + pSrc[n].im^2$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Phase

Computes the phase angles of elements of a complex vector.

Syntax

```
IppStatus ippsPhase_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPhase_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsPhase_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len, int
    scaleFactor);
IppStatus ippsPhase_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f* pDst,
    int len);
IppStatus ippsPhase_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f* pDst,
    int len);
IppStatus ippsPhase_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp32f* pDst,
    int len);
IppStatus ippsPhase_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp16s*
    pDst, int len, int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSrcRe</i>	Pointer to the source vector which stores the real components.
<i>pSrcIm</i>	Pointer to the source vector which stores the imaginary components.
<i>pDst</i>	Pointer to the vector which stores the phase (angle) components of the elements in radians. Phase values are in the range $(-\pi, \pi]$.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsPhase` is declared in the `ipps.h` file. This function returns the phase angles of elements of the complex input vector `pSrc`, or the complex input vector whose real and imaginary components are specified in the vectors `pSrcRe` and `pSrcIm`, respectively, and stores the result in the vector `pDst`. Phase values are returned in radians and are in the range $(-\pi, \pi]$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

PowerSpectr

Computes the power spectrum of a complex vector.

Syntax

```

IppStatus ippsPowerSpectr_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPowerSpectr_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPowerSpectr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsPowerSpectr_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPowerSpectr_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
    pDst, int len);
IppStatus ippsPowerSpectr_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
    pDst, int len);
IppStatus ippsPowerSpectr_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsPowerSpectr_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp32f*
    pDst, int len);

```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pSrcRe</code>	Pointer to the source vector which stores the real components.

<i>pSrcIm</i>	Pointer to the source vector which stores the imaginary components.
<i>pDst</i>	Pointer to the vector which stores the spectrum components of the elements.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsPowerSpectr` is declared in the `ipps.h` file. This function returns the power spectrum of the complex input vector *pSrc*, or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the results in the vector *pDst*. The power spectrum elements are squares of the magnitudes of the complex input vector elements:

$$pDst[n] = (pSrc[n].re)^2 + (pSrc[n].im)^2, \text{ or}$$

$$pDst[n] = (pSrcRe[n])^2 + (pSrcIm[n])^2.$$

To compute magnitudes, use the function [ippsMagnitude](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Real

Returns the real part of a complex vector in a second vector.

Syntax

```

IppStatus ippsReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, int len);
IppStatus ippsReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe, int len);
IppStatus ippsReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe, int len);

```


Parameters

<i>pSrc</i>	Pointer to the complex source vector.
<i>pDstRe</i>	Pointer to the destination vector with real parts.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsReal` is declared in the `ipps.h` file. This function returns the real part of the complex vector *pSrc* in the vector *pDstRe*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDstRe</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Imag

Returns the imaginary part of a complex vector in a second vector.

Syntax

```
IppStatus ippsImag_16sc(const Ipp16sc* pSrc, Ipp16s* pDstIm, int len);  
IppStatus ippsImag_32fc(const Ipp32fc* pSrc, Ipp32f* pDstIm, int len);  
IppStatus ippsImag_64fc(const Ipp64fc* pSrc, Ipp64f* pDstIm, int len);
```

Parameters

<i>pSrc</i>	Pointer to the complex source vector.
<i>pDstIm</i>	Pointer to the destination vector with imaginary parts.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsImag` is declared in the `ipps.h` file. This function returns the imaginary part of a complex vector *pSrc* in the vector *pDstIm*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDstIm</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

RealToCplx

Returns a complex vector constructed from the real and imaginary parts of two real vectors.

Syntax

```

IppStatus ippRealToCplx_16s(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp16sc*
    pDst, int len);
IppStatus ippRealToCplx_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32fc*
    pDst, int len);
IppStatus ippRealToCplx_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64fc*
    pDst, int len);

```

Parameters

<i>pSrcRe</i>	Pointer to the vector with real parts of complex elements.
<i>pSrcIm</i>	Pointer to the vector with imaginary parts of complex elements.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector.

Description

The function `ippRealToCplx` is declared in the `ipps.h` file. This function returns a complex vector *pDst* constructed from the real and imaginary parts of the input vectors *pSrcRe* and *pSrcIm*.

If *pSrcRe* is NULL, the real component of the vector is set to zero.

If *pSrcIm* is NULL, the imaginary component of the vector is set to zero.

Note that both pointers can not be NULL.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> pointer is NULL. The pointer <code>pSrcRe</code> or <code>pSrcIm</code> can be NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

CplxToReal

Returns the real and imaginary parts of a complex vector in two respective vectors.

Syntax

```
IppStatus ippCplxToReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, Ipp16s* pDstIm,
    int len);
IppStatus ippCplxToReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe, Ipp32f* pDstIm,
    int len);
IppStatus ippCplxToReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe, Ipp64f* pDstIm,
    int len);
```

Parameters

<code>pSrc</code>	Pointer to the complex vector <code>pSrc</code> .
<code>pDstRe</code>	Pointer to the output vector with real parts.
<code>pDstIm</code>	Pointer to the output vector with imaginary parts.
<code>len</code>	Number of elements in the vector.

Description

The function `ippCplxToReal` is declared in the `ipps.h` file. This function returns the real and imaginary parts of a complex vector `pSrc` in two vectors `pDstRe` and `pDstIm`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the data vector pointer is NULL.

`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0.

Threshold

Performs the threshold operation on the elements of a vector by limiting the element values by `level`.

Syntax

```

IppStatus ippThreshold_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s level,
    IppCmpOp relOp);
IppStatus ippThreshold_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f level,
    IppCmpOp relOp);
IppStatus ippThreshold_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f level,
    IppCmpOp relOp);
IppStatus ippThreshold_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, Ipp32f
    level, IppCmpOp relOp);
IppStatus ippThreshold_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, Ipp64f
    level, IppCmpOp relOp);
IppStatus ippThreshold_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, Ipp16s
    level, IppCmpOp relOp);
IppStatus ippThreshold_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level, IppCmpOp relOp);
IppStatus ippThreshold_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level, IppCmpOp relOp);
IppStatus ippThreshold_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level, IppCmpOp relOp);
IppStatus ippThreshold_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level, IppCmpOp
    relOp);
IppStatus ippThreshold_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level, IppCmpOp
    relOp);
IppStatus ippThreshold_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level, IppCmpOp
    relOp);

```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.

<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.				
<i>len</i>	Number of elements in the vector.				
<i>level</i>	Value used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> . This argument must always be real. For complex versions, it must be positive and represent magnitude.				
<i>relop</i>	Values of this argument specify which relational operator to use and whether <i>level</i> is an upper or lower bound for the input. The <i>relop</i> must have one of the following values: <table> <tr> <td><i>ippCmpLess</i></td><td>Specifies the “less than” operator and <i>level</i> is a lower bound.</td></tr> <tr> <td><i>ippCmpGreater</i></td><td>Specifies the “greater than” operator and <i>level</i> is an upper bound.</td></tr> </table>	<i>ippCmpLess</i>	Specifies the “less than” operator and <i>level</i> is a lower bound.	<i>ippCmpGreater</i>	Specifies the “greater than” operator and <i>level</i> is an upper bound.
<i>ippCmpLess</i>	Specifies the “less than” operator and <i>level</i> is a lower bound.				
<i>ippCmpGreater</i>	Specifies the “greater than” operator and <i>level</i> is an upper bound.				

Description

The function `ippsThreshold` is declared in the `ipps.h` file. This function performs the threshold operation on the vector *pSrc* by limiting each element by the threshold value *level*. Function operation is similar to that of the functions `ippsThreshold_LT`, `ippsThreshold_GT` but its interface contains the *relop* argument that specifies the type of the comparison operation to perform.

The in-place flavors of `ippsThreshold` perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

The *relop* argument specifies which relational operator to use: “greater than” or “less than,” and determines whether *level* is an upper or lower bound for the input, respectively. The formula for `ippsThreshold` called with the *ippCmpLess* flag is:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold`, the *level* argument is always real. The formula for complex `ippsThreshold` called with the *ippCmpLess* flag is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the function `ippsThreshold`. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the “less than” operation (with the `ippCmpLess` flag) the coordinates are rounded to the infinity (+Inf for positive coordinates, and -Inf for negative), and for the “greater than” operation (with the `ippCmpGreater` flag) the coordinates are rounded to 0. See [Example 5-13](#) for more information about using the “complex” function `ippsThreshold_16sc_I`.

[Example 5-12](#) shows how to use the “real” function `ippsThreshold_16s_I`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsThreshNegLevelErr</code>	Indicates an error when <i>level</i> for the complex version is negative (see appendix A “Handling of Special Cases” for more information).

Example 5-12 Using the Real `ippsThreshold` Function

```
IppStatus threshold( void ) {
    Ipp16s x[4] = { -1, 0, 2, 3 };
    IppStatus st = ippsThreshold_16s_I( x, 4, 2, ippCmpLess );
    printf_16s("threshold result =", x, 4, st );
    return st;
}
```

Output:

```
threshold result =  2 2 2 3
```

Example 5-13 Using the Complex ippsThreshold Function

```

IppStatus cmplx_threshold(void) {
    Ipp16sc x[4] = {{2,3}, {3,3}, {4,3}, {4,2}};
    /// level is near to the point {2,3} = 3.6
    /// the point {2,3} is to be replaced
    /// the computed coordinates are {2.2188,3.3282}
    /// the point used is {3,4};
    /// notice that it is the point with the phase,
    /// nearest to the source
    IppStatus st = ippsThreshold_16sc_I(x, 4, 4, ippCmpLess);
    printf_16sc("complex threshold result =", x, 4, st);
    return st;
}

```

Output:

```
complex threshold result = {3, 4} {3, 3} {4, 3} {4, 2}
```

Threshold_LT, Threshold_GT

Performs the threshold operation on the elements of a vector by limiting the element values by level.

Syntax

```

IppStatus ippsThreshold_LT_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s level);
IppStatus ippsThreshold_LT_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, Ipp32s level);
IppStatus ippsThreshold_LT_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, Ipp64f level);

```

```

IppStatus ippsThreshold_LT_16sc(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
    level);
IppStatus ippsThreshold_LT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_LT_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_LT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, Ipp32s level);
IppStatus ippsThreshold_GT_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, Ipp32f
    level);
IppStatus ippsThreshold_GT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, Ipp64f
    level);
IppStatus ippsThreshold_GT_16sc(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
    level);
IppStatus ippsThreshold_GT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_GT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc_I(Ipp16s* pSrcDst, int len, Ipp16s level);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

level Value used to limit each element of *pSrc* or *pSrcDst*. This argument must always be real. For complex versions, it must be positive and represent magnitude.

Description

The functions `ippsThreshold_LT`, `ippsThreshold_GT` are declared in the `ipps.h` file. They implement thresholding of the vector *pSrc* by limiting each element by the threshold value *level*. These functions perform the similar operation to the `ippsThreshold` function but are designed for the fixed type of the compare operation to use: `ippsThreshold_LT` is for the "less than" comparison, while `ippsThreshold_GT` is for the "greater than" comparison.

The in-place flavors perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

`ippsThreshold_LT`. The `ippsThreshold_LT` function performs the operation “less than”, and *level* is a lower bound for the input. The formula for `ippsThreshold_LT` is the following:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold_LT`, the *level* argument is always real.

The formula for complex `ippsThreshold_LT` is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

`ippsThreshold_GT`. The function `ippsThreshold_GT` performs the operation “greater than” and *level* is an upper bound for the input.

The formula for `ippsThreshold_GT` is the following:

$$pDst[n] = \begin{cases} level, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold_GT`, the *level* argument is always real.

The formula for complex `ippsThreshold_GT` is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the threshold functions. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the “less than” operation (the `ippsthreshold_LT` function) the coordinates are rounded to the infinity (+Inf for positive coordinates, and -Inf for negative), and for the “greater than” operation (the `ippsthreshold_GT` function) the coordinates are rounded to 0.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0
<code>ippStsThreshNegLevelErr</code>	Indicates an error when <i>level</i> for the complex version is negative (see appendix A “Handling of Special Cases” for more information).

Threshold_LTAbs, Threshold_GTAbs

Performs the threshold operation on the absolute values of elements of a vector.

Syntax

```
IppStatus ippsthreshold_LTAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s level);
IppStatus ippsthreshold_LTAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    Ipp32s level);
```

```

IppStatus ippsThreshold_LTAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_LTAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_LTAbs_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_LTAbs_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_LTAbs_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTAbs_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);

IppStatus ippsThreshold_GTAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s level);
IppStatus ippsThreshold_GTAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    Ipp32s level);
IppStatus ippsThreshold_GTAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_GTAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_GTAbs_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GTAbs_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_GTAbs_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GTAbs_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>level</i>	Value used to limit each element of source vector. This argument can not be negative.

Description

The functions `ippsThreshold_LTAbs`, `ippsThreshold_GTAbs` are declared in the `ipps.h` file. They implement thresholding of the vector *pSrc* by limiting absolute value of each element by the threshold value *level*. These functions perform the compare operation of the fixed type:

`ippsThreshold_LTAbs` is for the "less than" comparison, while `ippsThreshold_GTAbs` is for the "greater than" comparison. Elements of the result vector *pDst* have the same sign that the source elements.

The in-place flavors perform the threshold operation on the vector *pSrcDst*.

ippsThreshold_LTAbs. The `ippsThreshold_LTAbs` function performs the operation “less than”, and *level* is a lower bound for the input. The formula for `ippsThreshold_LTAbs` is the following:

$$pDst[n] = \begin{cases} level & \text{if } abs(pSrc[n]) < level, & pSrc[n] \geq 0 \\ -level & \text{if } abs(pSrc[n]) < level, & pSrc[n] < 0 \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_GTAbs. The function `ippsThreshold_GTAbs` performs the operation “greater than” and *level* is an upper bound for the input. The formula for `ippsThreshold_GTAbs` is the following:

$$pDst[n] = \begin{cases} level & \text{if } abs(pSrc[n]) > level, & pSrc[n] \geq 0 \\ -level & \text{if } abs(pSrc[n]) > level, & pSrc[n] < 0 \\ pSrc[n], & otherwise \end{cases}$$

[Example 5-14](#) shows how to use the function `ippsThreshold_LTAbs_64f_I`.

Example 5-14 Using the `ippThreshold_LTAbs` function

```

IppStatus thresholdLTAbs( void ) {
    Ipp64f      vec[7] = { -4.01, -4.0, -3.9, 0.0, 2.5, 4.0, 4.5 };
    Ipp64f      level = 4.0;
    IppStatus    st = ippThreshold_LTAbs_64f_I( vec, 7, level );
    printf("threshold abs    =   %f  %f  %f  %f\n",
           vec[0],vec[1],vec[2],vec[3]);
    printf("                  %f  %f  %f\n",
           vec[4],vec[5],vec[6]);
    return st;
}

```

Output:

```

threshold abs = -4.010000 -4.000000 -4.000000 4.000000
                4.000000 4.000000 4.500000

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.
<code>ippStsThreshNegLevelErr</code>	Indicates an error if <i>level</i> is negative.

Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal

Performs the threshold operation on the elements of a vector by limiting the element values by level.

Syntax

```

IppStatus ippsThreshold_LTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
    level, Ipp16s value);
IppStatus ippsThreshold_LTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
    level, Ipp32f value);
IppStatus ippsThreshold_LTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f
    level, Ipp64f value);
IppStatus ippsThreshold_LTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_LTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_LTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_LTVal_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level, Ipp16s
    value);
IppStatus ippsThreshold_LTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level, Ipp32f
    value);
IppStatus ippsThreshold_LTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level, Ipp64f
    value);
IppStatus ippsThreshold_LTVal_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level, Ipp16sc
    value);
IppStatus ippsThreshold_LTVal_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level, Ipp32fc
    value);
IppStatus ippsThreshold_LTVal_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level, Ipp64fc
    value);
IppStatus ippsThreshold_GTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
    level, Ipp16s value);
IppStatus ippsThreshold_GTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
    level, Ipp32f value);

```

```
IppStatus ippsThreshold_GTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_GTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_GTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_GTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_GTVal_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level,
    Ipp16s value);
IppStatus ippsThreshold_GTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level,
    Ipp32f value);
IppStatus ippsThreshold_GTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level,
    Ipp64f value);
IppStatus ippsThreshold_GTVal_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level,
    Ipp16sc value);
IppStatus ippsThreshold_GTVal_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level,
    Ipp32fc value);
IppStatus ippsThreshold_GTVal_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level,
    Ipp64fc value);
IppStatus ippsThreshold_LTValGTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s levelLT, Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);
IppStatus ippsThreshold_LTValGTVal_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    Ipp32s levelLT, Ipp32s valueLT, Ipp32s levelGT, Ipp32s valueGT);
IppStatus ippsThreshold_LTValGTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f levelLT, Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);
IppStatus ippsThreshold_LTValGTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f levelLT, Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);
IppStatus ippsThreshold_LTValGTVal_16s_I(Ipp16s* pSrcDst, int len, Ipp16s levelLT,
    Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);
IppStatus ippsThreshold_LTValGTVal_32s_I(Ipp32s* pSrcDst, int len, Ipp32s levelLT,
    Ipp32s valueLT, Ipp32s levelGT, Ipp32s valueGT);
IppStatus ippsThreshold_LTValGTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f levelLT,
    Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);
IppStatus ippsThreshold_LTValGTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f levelLT,
    Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>level</i>	Value used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> . This argument must always be real. For complex versions, it must be positive and represent magnitude.
<i>levelLT</i>	Low bound used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> for the <code>ippsThreshold_LTVaLGTVal</code> function.
<i>levelGT</i>	Upper bound used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> for the <code>ippsThreshold_LTVaLGTVal</code> function.
<i>value</i>	Value to be assigned to vector elements which are “less than” or “greater than” <i>level</i> .
<i>valueLT</i>	Value to be assigned to vector elements which are less than <i>levelLT</i> for the <code>ippsThreshold_LTVaLGTVal</code> function.
<i>valueGT</i>	Value to be assigned to vector elements which are greater than <i>levelGT</i> for the <code>ippsThreshold_LTVaLGTVal</code> function.

Description

These functions are declared in the `ipps.h` file. They perform the threshold operation on the vector *pSrc* by limiting each element by the threshold value.

The in-place flavors of the function perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value.

ippsThreshold_LTVaL. The `ippsThreshold_LTVaL` function performs the operation “less than” and *level* is a lower bound for the input. The vector elements less than *level* are set to *value*.

The formula for `ippsThreshold_LTVaL` is:

$$pDst[n] = \begin{cases} value, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold_LTVAl`, the `level` argument is always real.

The formula for complex `ippsThreshold_LTVAl` is:

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

`ippsThreshold_GTVAl`. The `ippsThreshold_GTVAl` function performs the operation “greater than” and `level` is an upper bound for the input. The vector elements greater than `level` are set to `value`.

The formula for `ippsThreshold_GTVAl` is:

$$pDst[n] = \begin{cases} value, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold_GTVAl`, the `level` argument is always real.

The formula for complex `ippsThreshold_GTVAl` is:

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

`ippsThreshold_LTVAlGTVAl`. The `ippsThreshold_LTVAlGTVAl` function checks both the “less than” and “greater than” conditions. Argument `levelLT` is a lower bound and argument `levelGT` is an upper bound for the input. The source vector elements less than `levelLT` are set to `valueLT`, and the source vector elements greater than `levelGT` are set to `valueGT`. The value of `levelLT` should be less than or equal to `levelGT`.

The formula for `ippsThreshold_LTVAlGTVAl` is:

$$pDst[n] = \begin{cases} valueLT, & pSrc[n] < levelLT \\ pSrc[n], & levelLT \leq pSrc[n] \leq levelGT \\ valueGT, & pSrc[n] > levelGT \end{cases}$$

Application Notes

For all complex versions, `level` must be positive and represent a magnitude.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsThresholdErr</code>	Indicates an error when <i>levelLT</i> is greater than <i>levelGT</i> .
<code>ippStsThreshNegLevelErr</code>	Indicates an error when <i>level</i> for the complex version is negative (see appendix A “Handling of Special Cases” for more information).

Threshold_LTInv

Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.

Syntax

```

IppStatus ippsthreshold_LTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level);
IppStatus ippsthreshold_LTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level);
IppStatus ippsthreshold_LTInv_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level);
IppStatus ippsthreshold_LTInv_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level);
IppStatus ippsthreshold_LTInv_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsthreshold_LTInv_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsthreshold_LTInv_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsthreshold_LTInv_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.

<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>level</i>	Value used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> . This argument must always be real and positive.

Description

The function `ippsThreshold_LTInv` is declared in the `ipps.h` file. This function computes the inverse of elements of the vector *pSrc* and stores the result in *pDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The in-place flavors of `ippsThreshold_LTInv` compute the inverse of elements of the vector *pSrcDst* and store the result in *pSrcDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The threshold operation is performed to avoid division by zero. Since *level* represents a magnitude, it is always real and must be positive. The formula for `ippsThreshold_LTInv` is the following:

$$pDst[n] = \begin{cases} \frac{1}{level}, & abs(pSrc[n]) = 0 \\ \frac{abs(pSrc[n])}{pSrc[n] \cdot level}, & 0 < abs(pSrc[n]) < level \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

If the function encounters zero-valued vector elements and *level* is also 0 (see appendix A [“Handling of Special Cases”](#)), the output value is set to `Inf` (infinity), but operation execution is not aborted:

$$pDst[n] = \begin{cases} Inf, & pSrc[n] = 0 \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

[Example 5-15](#) shows how to use the function `ippsThreshold_LTInv_32f_I`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsThreshNegLevelErr</code>	Indicates an error when <i>level</i> is negative.
<code>ippStsInvZero</code>	Indicates a warning when <i>level</i> and a vector element are equal to 0. Operation execution is not aborted. The value of the destination vector element is <code>Inf</code> .

Example 5-15 Using the `ippsThreshold_LTInv` Function

```
IppStatus invThreshold(void) {
    Ipp32f x[4] = {-1, 0, 2, 3};
    IppStatus st = ippsThreshold_LTInv_32f_I(x, 4, 0);
    printf_32f("inv threshold =", x, 4, st);
    return st;
}
```

Output:

```
-- warning 4, INF result. Zero value met by invThreshold with zero level
inv threshold = -1.000000 1.#INF00 0.500000 0.333333
```

CartToPolar

Converts the elements of a complex vector to polar coordinate form.

Syntax

```
IppStatus ippsCartToPolar_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstMagn, Ipp32f* pDstPhase, int len);
IppStatus ippsCartToPolar_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstMagn, Ipp64f* pDstPhase, int len);
```

```

IppStatus ippsCartToPolar_32fc(const Ipp32fc* pSrc, Ipp32f* pDstMagn, Ipp32f*
    pDstPhase, int len);
IppStatus ippsCartToPolar_64fc(const Ipp64fc* pSrc, Ipp64f* pDstMagn, Ipp64f*
    pDstPhase, int len);
IppStatus ippsCartToPolar_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDstMagn,
    Ipp16s* pDstPhase, int len, int magnScaleFactor, int phaseScaleFactor);

```

<i>pSrc</i>	Pointer to the source vector.
<i>pSrcRe</i>	Pointer to the source vector which stores the real components of Cartesian X,Y pairs.
<i>pSrcIm</i>	Pointer to the source vector which stores the imaginary components of Cartesian X,Y pairs.
<i>pDstMagn</i>	Pointer to the vector which stores the magnitude (radius) component of the elements of the vector <i>pSrc</i> .
<i>pDstPhase</i>	Pointer to the vector which stores the phase (angle) component of the elements of the vector <i>pSrc</i> in radians. Phase values are in the range $(-\pi, \pi]$.
<i>len</i>	Number of elements in the vector.
<i>magnScaleFactor</i>	Integer scale factor for the magnitude component. Refer to Integer Scaling in Chapter 2.
<i>phaseScaleFactor</i>	Integer scale factor for the phase component. Refer to Integer Scaling .

Description

The function `ippsCartToPolar` is declared in the `ipps.h` file. This function converts the elements of a complex input vector *pSrc* or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, to polar coordinate form, and stores the magnitude (radius) component of each element in the vector *pDstMagn* and the phase (angle) component of each element in the vector *pDstPhase*.

[Example 5-16](#) verifies that points are lying in the unit radius circle.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Example 5-16 Using the ippsCartToPolar Function

```

IppStatus cart2polar( void ) {
    Ipp64f cart[6], magn[4], phase[4];
    int n;
    for (n=0; n<6; ++n) cart[n] = sin(IPP_2PI * n / 8);
    IppStatus st = ippsCartToPolar_64f( cart, cart+2, magn, phase, 4 );
    printf_64f( "magn =", magn, 4, st );
    return st;
}

```

Output:

```
magn = 1.000000 1.000000 1.000000 1.000000
```

PolarToCart

Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.

Syntax

```

IppStatus ippsPolarToCart_32f(const Ipp32f* pSrcMagn, const Ipp32f* pSrcPhase,
    Ipp32f* pDstRe, Ipp32f* pDstIm, int len);
IppStatus ippsPolarToCart_64f(const Ipp64f* pSrcMagn, const Ipp64f* pSrcPhase,
    Ipp64f* pDstRe, Ipp64f* pDstIm, int len);
IppStatus ippsPolarToCart_32fc(const Ipp32f* pSrcMagn, const Ipp32f* pSrcPhase,
    Ipp32fc* pDst, int len);
IppStatus ippsPolarToCart_64fc(const Ipp64f* pSrcMagn, const Ipp64f* pSrcPhase,
    Ipp64fc* pDst, int len);
IppStatus ippsPolarToCart_16sc_Sfs(const Ipp16s* pSrcMagn, const Ipp16s*
    pSrcPhase, Ipp16sc* pDst, int len, int magnScaleFactor, int
    phaseScaleFactor);

```

Parameters

<i>pSrcMagn</i>	Pointer to the source vector which stores the magnitude (radius) components of the elements in polar coordinate form.
<i>pSrcPhase</i>	Pointer to the vector which stores the phase (angle) components of the elements in polar coordinate form in radians. Phase values are in the range $(-\pi, \pi]$.
<i>pDst</i>	Pointer to the resulting vector which stores the complex pairs in Cartesian coordinates ($X + iY$).
<i>pDstRe</i>	Pointer to the resulting vector which stores the real components of Cartesian X,Y pairs.
<i>pDstIm</i>	Pointer to the resulting vector which stores the imaginary components of Cartesian X,Y pairs.
<i>len</i>	Number of elements in the vectors.
<i>magnScaleFactor</i>	Integer scale factor for the magnitude component. Refer to Integer Scaling in Chapter 2.
<i>phaseScaleFactor</i>	Integer scale factor for the phase component. Refer to Integer Scaling .

Description

The function `ippsPolarToCart` is declared in the `ipps.h` file. This function converts the polar form magnitude/phase pairs stored in the input vectors *pSrcMagn* and *pSrcPhase* into a complex vector and stores the results in the vector *pDst*, or stores the real components of the result in the vector *pDstRe* and the imaginary components in the vector *pDstIm*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

MaxOrder

Computes the maximum order of a vector.

Syntax

```
IppStatus ippsMaxOrder_16s(const Ipp16s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32s(const Ipp32s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32f(const Ipp32f* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_64f(const Ipp64f* pSrc, int len, int* pOrder);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>len</i>	Number of elements in the vector.
<i>pOrder</i>	Pointer to the result value.

Description

The function `ippsMaxOrder` is declared in the `ipps.h` file. This function finds the maximum binary number in elements of the exponent vector *pSrc*, and stores the result in *pOrder*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pOrder</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsNanArg</code>	Indicates a warning when NaN is encountered in the input data vector.

Preemphasize

Computes preemphasis of a single precision real signal in-place.

Syntax

```
IppStatus ippsPreemphasize_16s(Ipp16s* pSrcDst, int len, Ipp32f val);  
IppStatus ippsPreemphasize_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
```

Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>val</i>	Multiplier factor used in the difference signal preemphasis equation.

Description

The in-place function `ippsPreemphasize` is declared in the `ipps.h` file. This function computes preemphasis of a real signal *pSrcDst*. The computation is performed according to the difference signal preemphasis equation:

$$y(n) = x(n) - val \cdot x(n - 1),$$

where $y(n)$ is the preemphasized output, $x(n)$ is the input, and *val* is the multiplier factor.

Note that usually $val = 0.95$ for speech signals.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Flip

Reverses the order of elements in a vector.

Syntax

```
IppStatus ippsFlip_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsFlip_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsFlip_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsFlip_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsFlip_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsFlip_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsFlip_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsFlip_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsFlip` is declared in the `ipps.h` file. This function stores the elements of a source vector *pSrc* to a destination vector *pDst* in reverse order according to the following formula:

$$pDst[n] = pSrc[len-n-1], \quad n=0..len-1$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

FindNearestOne

Finds an element of the table which is closest to the specified value.

Syntax

```
IppStatus ippsFindNearestOne_16u(Ipp16u inpVal, Ipp16u* pOutVal, int* pOutIndex,
    const Ipp16u *pTable, int tblLen);
```

Parameters

<i>inpVal</i>	Reference value.
<i>pOutVal</i>	Pointer to the output value.
<i>pOutIndex</i>	Pointer to the output index.
<i>pTable</i>	Pointer to the table for searching.
<i>tblLen</i>	Number of elements in the table.

Description

The function `ippsFindNearestOne` is declared in the `ipps.h` file. This function searches through the table *pTable* for an element which is closest to the specified reference value *inpVal*. The resulting element and its index are stored in *pOutVal* and *pOutIndex*, respectively. The table elements should satisfy the condition $pTable[i] \leq pTable[i+1]$. The function uses the following distance criterion for determining the closest element: $\min(|inpVal - pTable[i]|)$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>tblLen</i> is less than or equal to 0.

FindNearest

Finds table elements that are closest to the elements of the specified vector.

Syntax

```
IppStatus ippsFindNearest_16u(const Ipp16u* pVals, Ipp16u* pOutVals, int* pOutIndexes,
    int len, const Ipp16u *pTable, int tblLen);
```

Parameters

<i>pVals</i>	Pointer to the vector containing reference values.
<i>pOutVals</i>	Pointer to the output vector.
<i>pOutIndexes</i>	Pointer to the array that stores output indexes.
<i>len</i>	Number of elements in the input vector.
<i>pTable</i>	Pointer to the table for searching.
<i>tblLen</i>	Number of elements in the table.

Description

The function `ippsFindNearest` is declared in the `ipps.h` file. This function searches through the table *pTable* for elements which are closest to the reference elements of the input vector *pVals*. The resulting elements and their indexes are stored in *pOutVals* and *pOutIndexes*, respectively.

The table elements should satisfy the condition $pTable[i] \leq pTable[i+1]$.

The function uses the following distance criterion for determining the table element closest to *pVal[k]*:

$\min(|pVals[k] - pTable[i]|)$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>tblLen</i> or <i>len</i> is less than or equal to 0.

Viterbi Decoder Functions

This section describes the functions that perform operations of Viterbi decoding in the V34 receiver. The encoding is performed in accordance with the algorithm that is described in the section 9.6.3 of the ITU-T Recommendation V.34 (see [[ITUV34](#)]).

GetVarPointDV

Fills the array with the information about points that are closest to the received point.

Syntax

```
IppStatus ippsGetVarPointDV_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, Ipp16sc*
    pVariantPoint, const Ipp8u* pLabel, int state);
```

Parameters

<i>pSrc</i>	Pointer to the reference point in format 9:7
<i>pDst</i>	Pointer to the closest to the reference point left and bottom complex point in format 9:7
<i>pVariantPoint</i>	Pointer to the array where the point information is stored
<i>pLabel</i>	Pointer to the table that stores the labels.
<i>state</i>	Number of states of a convolution coder.

Description

The function `ippsGetVarPointDV` is declared in the `ipps.h` file. This function fills the specified array *pVariantPoint* with the information about 2D complex points that are closest to the reference point *pSrc*. This information includes the corresponding labels from the offset table and calculated errors. The number of possible states may be 16, 32, and 64. The number of points in the array depends on the number of states: if *state* = 16, then 4 points will be referenced in the array, if *state* = 32 or 64, then 8 points will be referenced.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .

CalcStatesDV

Calculates the states of the Viterbi decoder.

Syntax

```
IppStatus ippsCalcStatesDV_16sc(const Ipp16u* pathError, const Ipp8u* pNextState,
    Ipp16u* pBranchError, const Ipp16s* pCurrentSubsetPoint, Ipp16s* pPathTable,
    int state, int presentIndex);
```

Parameters

<i>pPathError</i>	Pointer to the table of path error metrics.
<i>pNextState</i>	Pointer to the next state table.
<i>pBranchError</i>	Pointer to the branch error table.
<i>pCurrentSubsetPoint</i>	Pointer to the current 4D subset.
<i>pPathTable</i>	Pointer to the Viterbi path table.
<i>state</i>	Number of states of a convolutional encoder.
<i>presentIndex</i>	Start index in Viterbi path table.

Description

The function `ippsCalcStatesDV` is declared in the `ipps.h` file. This function computes the possible states of the Viterbi decoder and fills the table of the accumulated errors *pPathError* and the working Viterbi path table *pPathTable*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is NULL.

BuildSymblTableDV4D

Fills the array with the information about possible 4D symbols.

Syntax

```
IppStatus ippsBuildSymblTableDV4D_16sc(const Ipp16sc* pVariantPoint, Ipp16sc*
    pCurrentSubsetPoint, int state, int bitInversion);
```

Parameters

<i>pVariantPoint</i>	Pointer to the array of possible 2D points.
<i>pCurrentSubsetPoint</i>	Pointer to the array with information about possible 4D symbols.
<i>state</i>	Number of states of a convolutional encoder.
<i>bitInversion</i>	Bit inversion.

Description

The function `ippsBuildSymblTableDV4D` is declared in the `ipps.h` file. This function fills the array `pCurrentSubsetPoint` with information about possible 4D symbols.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is NULL.

UpdatePathMetricsDV

Searches for the state with the minimum path metric.

Syntax

```
IppStatus ippsUpdatePathMetricsDV_16u(Ipp16u* pBranchError, Ipp16u* pMinPathError,
    Ipp8u* pMinSost, Ipp16u* pPathError, int state);
```

Parameters

<i>pBranchError</i>	Pointer to the branch error table.
<i>pMinPathError</i>	Pointer to the current minimum path error metric.
<i>pMinSost</i>	Pointer to the number of states with the minimum metric.
<i>pPathError</i>	Pointer to the table of accumulated path errors.
<i>state</i>	Number of states of a convolutional encoder.

Description

The function `ippBuildSymbTableDV4D` is declared in the `ipp.h` file. This function searches for the state with the minimum path error metric and stores its number in the *pMinSost*. For all states, the minimum metric is subtracted from the path metric. The branch errors for all states are set to be infinitely large as is required for the next step of the Viterbi decoding.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .

Windowing Functions

This chapter describes several of the windowing functions commonly used in signal processing. A window is a mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

Understanding Window Functions

The Intel IPP provides the following functions to generate window samples:

- Bartlett windowing function
- Blackman family of windowing functions
- Hamming windowing function
- Hann windowing function
- Kaiser windowing function

These functions generate the window samples and multiply them into an existing signal. To obtain the window samples themselves, initialize the vector argument to the unity vector before calling the window function.

If you want to multiply different frames of a signal by the same window multiple times, it is better to first calculate the window by calling one of the windowing functions (`ippsWinHann`, for example) on a vector with all elements set to 1.0. Then use one of the vector multiplication functions (`ippsMul`, for example) to multiply the window into the signal each time a new set of input samples is available. This avoids repeatedly calculating the window samples. This is illustrated in [Example 5-17](#).

Example 5-17 Window and FFT Many Frames of a Signal

```
void multiFrameWin( void ) {
    Ipp32f win[LEN], x[LEN], X[LEN];
    IppsFFTSpec_R_32f* ctx;
    ippsSet_32f( 1, win, LEN );
    ippsWinHann_32f_I( win, LEN );
    /// ... initialize FFT context
    while(1 ){
        /// ... get x signal
        ///
        ippsMul_32f_I( win, x, LEN );
        ippsFFTFwd_RToPack_32f( x, X, ctx, 0 );
    }
}
```

Related Topics

For more information on windowing, see: [\[Jack89\]](#), section 7.3, *Windows in Spectrum Analysis*; [\[Jack89\]](#), section 9.1, *Window-Function Technique*; and [\[Mit93\]](#), section 16-2, *Fourier Analysis of Finite-Time Signals*.

For more information on these references, see also the [Bibliography](#) at the end of this manual.

WinBartlett

Multiplies a vector by a Bartlett windowing function.

Syntax

```
IppStatus ippsWinBartlett_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBartlett_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBartlett_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBartlett_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinBartlett_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinBartlett_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinBartlett_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBartlett_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBartlett_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBartlett_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBartlett_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBartlett_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsWinBartlett` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Bartlett (triangle) window, and stores the result in *pDst*.

The in-place flavors of `ippsWinBartlett` multiply the *pSrcDst* by the Bartlett (triangle) window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window.

The Bartlett window is defined as follows:

$$w_{\text{bartlett}}(n) = \begin{cases} \frac{2n}{len-1}, & 0 \leq n \leq \frac{len-1}{2} \\ 2 - \frac{2n}{len-1}, & \frac{len-1}{2} < n \leq len-1 \end{cases}$$

[Example 5-18](#) shows how to use the function `ippsWinBartlett_32f_I`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than 3.

Example 5-18 Using the `ippsWinBartlett` Function

```
void bartlett(void) {
    Ipp32f x[8];
    ippsSet_32f(1, x, 8);
    ippsWinBartlett_32f_I(x, 8);
    printf_32f("bartlett (half) =", x, 4, ippStsNoErr);
}

Output:
    bartlett (half) =  0.000000 0.285714 0.571429 0.857143

Matlab* Analog:
    >> b = bartlett(8); b(1:4)'
```

WinBlackman

Multiplies a vector by a Blackman windowing function.

Syntax

```
IppStatus ippsWinBlackmanQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, int
    alphaQ15);
```

```

IppStatus ippsWinBlackmanQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, int
    alphaQ15);
IppStatus ippsWinBlackman_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, float alpha);
IppStatus ippsWinBlackman_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, float
    alpha);
IppStatus ippsWinBlackman_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, float alpha);
IppStatus ippsWinBlackman_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, float
    alpha);
IppStatus ippsWinBlackman_64f(Ipp64f* pSrc, Ipp64f* pDst, int len, double alpha);
IppStatus ippsWinBlackman_64fc(Ipp64fc* pSrc, Ipp64fc* pDst, int len, double alpha);
IppStatus ippsWinBlackmanQ15_16s_I(Ipp16s* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinBlackmanQ15_16s_ISfs(Ipp16s* pSrcDst, int len, int alphaQ15, int
    scaleFactor);
IppStatus ippsWinBlackmanQ15_16sc_I(Ipp16sc* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinBlackman_16s_I(Ipp16s* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_16sc_I(Ipp16sc* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_32f_I(Ipp32f* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_32fc_I(Ipp32fc* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_64f_I(Ipp64f* pSrcDst, int len, double alpha);
IppStatus ippsWinBlackman_64fc_I(Ipp64fc* pSrcDst, int len, double alpha);
IppStatus ippsWinBlackmanStd_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBlackmanStd_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinBlackmanStd_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBlackmanStd_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinBlackmanStd_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBlackmanStd_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinBlackmanStd_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBlackmanOpt_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);

```

```

IppStatus ippsWinBlackmanOpt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBlackmanOpt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinBlackmanOpt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBlackmanOpt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinBlackmanOpt_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64fc_I(Ipp64fc* pSrcDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>alpha</i>	Adjustable parameter associated with the Blackman windowing equation.
<i>alphaQ15</i>	Scaled version of <i>alpha</i> . The <i>scaleFactor</i> value is 15.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The `ippsWinBlackman` family of functions are declared in the `ipps.h` file. These functions multiply the vector *pSrc* by the Blackman window, and store the result in *pDst*.

The in-place flavors of `ippsWinBlackman` multiply the vector *pSrcDst* by the Blackman window, and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The functions for the Blackman family of windows are defined below.

ippsWinBlackman. The function `ippsWinBlackman` allows the application to specify *alpha*. The Blackman window is defined as follows:

$$w_{blackman}(n) = \frac{\alpha + 1}{2} - 0.5 \cos\left(\frac{2\pi n}{len - 1}\right) - \frac{\alpha}{2} \cos\left(\frac{4\pi n}{len - 1}\right)$$

ippWinBlackmanQ15. The function `ippWinBlackmanQ15` multiplies a vector by a Blackman window with `alphaQ15` scaled according to the factor 15.

ippWinBlackmanStd. The standard Blackman window is provided by the function `ippWinBlackmanStd`, which simply multiplies a vector by a Blackman window with the standard value of `alpha` shown below:

$$\alpha = -0.16$$

ippWinBlackmanOpt. The function `ippWinBlackmanOpt` provides a modified window that has a 30 dB/octave roll-off by multiplying a vector by a Blackman window with the optimal value of `alpha` shown below:

$$\alpha = -\frac{0.5}{1 + \cos\frac{2\pi}{len - 1}}$$

The minimum `len` is equal to 4. For large `len`, the optimal `alpha` converges asymptotically to the asymptotic `alpha`; the application can use the asymptotic value of `alpha` shown below:

$$\alpha = -0.25$$

[Example 5-19](#) shows how to use the function `ippWinBlackmanStd_32f_I`

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than 4 for the function <code>ippWinBlackmanOpt</code> and less than 3 for all other functions of the family.

Example 5-19 Using the ippsWinBlackmanStd Function

```
void blackman(void) {
    Ipp32f x[8];
    ippsSet_32f(1, x, 8);
    ippsWinBlackmanStd_32f_I(x, 8);
    printf_32f("blackman (half) =", x, 4, ippsStsNoErr);
}
```

Output:

```
blackman(half) = 0.000000 0.090453 0.459183 0.920364
```

Matlab* Analog:

```
>> b = blackman(8)'; b(1:4)
```

WinHamming

Multiplies a vector by a Hamming windowing function.

Syntax

```
IppStatus ippsWinHamming_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHamming_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHamming_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHamming_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHamming_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHamming_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHamming_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHamming_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHamming_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHamming_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHamming_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHamming_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsWinHamming` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Hamming window and stores the result in *pDst*.

The in-place flavors of `ippsWinHamming` multiply the vector *pSrcDst* by the Hamming window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hamming window is defined as follows:

$$w_{\text{hamming}}(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{len - 1}\right)$$

[Example 5-20](#) shows how to use the function `ippsWinHamming_32f_I`

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than 3.

Example 5-20 Using the ippsWinHamming Function

```
void hamming(void) {
    Ipp32f x[8];
    ippsSet_32f(1, x, 8);
    ippsWinHamming_32f_I(x, 8);
    printf_32f("hamming(half) =", x, 4, ippStsNoErr);
}

Output:
    hamming(half) =  0.080000 0.253195 0.642360 0.954446

Matlab* Analog:
    >> b = hamming(8); b(1:4)'
```

WinHann

Multiplies a vector by a Hann windowing function.

Syntax

```
IppStatus ippsWinHann_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHann_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHann_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHann_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHann_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHann_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHann_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHann_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHann_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHann_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHann_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHann_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsWinHann` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Hann window and stores the result in *pDst*.

The in-place flavors of `ippsWinHann` multiply the vector *pSrcDst* by the Hann window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hann window is defined as follows:

$$w_{hann}(n) = 0.5 - 0.5 \cos\left(\frac{2\pi n}{len - 1}\right)$$

[Example 5-21](#) shows how to use the function `ippsWinHann_32f_I`

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than 3.

Example 5-21 Using the ippsWinHann Function

```
void hann(void) {
    Ipp32f x[8];
    ippsSet_32f(1, x, 8);
    ippsWinHann_32f_I(x, 8);
    printf_32f("hann(half) =", x, 4, ippStsNoErr);
}
```

Output:

```
hann(half) = 0.000000 0.188255 0.611260 0.950484
```

Matlab* Analog:

```
>> N = 8; n = 0:N-1; 0.5*(1-cos(2*pi*n/(N-1)))
```

WinKaiser

Multiplies a vector by a Kaiser windowing function.

Syntax

```
IppStatus ippsWinKaiser_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, float alpha);
IppStatus ippsWinKaiser_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, float alpha);
IppStatus ippsWinKaiser_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, float alpha);
IppStatus ippsWinKaiser_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, float alpha);
IppStatus ippsWinKaiser_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, float alpha);
IppStatus ippsWinKaiser_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, float alpha);
IppStatus ippsWinKaiserQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, int alphaQ15);
IppStatus ippsWinKaiserQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, int
    alphaQ15);

IppStatus ippsWinKaiser_16s_I(Ipp16s* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32f_I(Ipp32f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_64f_I(Ipp64f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_16sc_I(Ipp16sc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32fc_I(Ipp32fc* pSrcDst, int len, float alpha);
```

```
IppStatus ippsWinKaiser_64fc_I(Ipp64fc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiserQ15_16s_I(Ipp16s* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinKaiserQ15_16sc_I(Ipp16sc* pSrcDst, int len, int alphaQ15);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>alpha</i>	Adjustable parameter associated with the Kaiser windowing equation.
<i>alphaQ15</i>	Scaled version of <i>alpha</i> . The <i>scaleFactor</i> value is 15.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsWinKaiser` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Kaiser window, and stores the result in *pDst*.

The in-place flavors of `ippsWinKaiser` multiply the vector *pSrcDst* by the Kaiser window and store the result in *pSrcDst*.

ippsWinKaiser. The function `ippsWinKaiser` allows the application to specify *alpha*. The function multiplies both real and imaginary parts of the complex vector by the same window. The Kaiser family of windows are defined as follows:

$$w_{kaiser}(n) = \frac{I_0\left(\alpha \sqrt{\left(\frac{len-1}{2}\right)^2 - \left(n - \left(\frac{len-1}{2}\right)\right)^2}\right)}{I_0\left(\alpha \left(\frac{len-1}{2}\right)\right)}$$

Here $I_0()$ is the modified zero-order Bessel function of the first kind.

ippsWinKaiserQ15. The function `ippsWinKaiserQ15` multiplies a vector by a Kaiser window with *alphaQ15* scaled according to the factor 15.

[Example 5-22](#) shows how to use the function `ippsWinKaiser_32f_I`

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> , <i>pSrc</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than 1.
<code>ippStsHugeWinErr</code>	Indicates an error when the Kaiser window is too big.

Example 5-22 Using the `ippsWinKaiser` Function

```
void kaiser(void) {
    Ipp32f x[8];
    IppStatus st;
    ippsSet_32f(1, x, 8);
    st = ippsWinKaiser_32f_I( x, 8, 1.0f );
    printf_32f("kaiser(half) =", x, 4, ippStsNoErr);
}
```

Output:

```
kaiser(half) = 0.135534 0.429046 0.755146 0.970290
```

Matlab* Analog:

```
>> kaiser(8,7/2)'
```

Statistical Functions

This section describes the Intel IPP functions that compute the vector measure values: maximum, minimum, mean, and standard deviation.

Sum

Computes the sum of the elements of a vector.

Syntax

```
IppStatus ippsSum_32f( const Ipp32f* pSrc, int len, Ipp32f* pSum, IppHintAlgorithm hint );
IppStatus ippsSum_32fc( const Ipp32fc* pSrc, int len, Ipp32fc* pSum, IppHintAlgorithm hint );
```

```

IppStatus ippsSum_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSum_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pSum);
IppStatus ippsSum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pSum, int scaleFactor);
IppStatus ippsSum_32s_Sfs(const Ipp32s* pSrc, int len, Ipp32s* pSum, int scaleFactor);
IppStatus ippsSum_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pSum, int
    scaleFactor);
IppStatus ippsSum_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pSum, int
    scaleFactor);
IppStatus ippsSum_16sc32sc_Sfs(const Ipp16sc* pSrc, int len, Ipp32sc* pSum, int
    scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSum</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>hint</i>	Suggests using specific code. The values for the <i>hint</i> argument are described in Flag and Hint Arguments .
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSum` is declared in the `ipps.h` file. This function computes the sum of the elements of the vector *pSrc* and stores the result in *pSum*.

The sum of the elements of *pSrc* is defined by the formula:

$$sum = \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

When computing the sum of integer numbers, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. The scaling is performed in accordance with the *scaleFactor* value.

[Example 5-23](#) shows how to use the function `ippsSum`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSum</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Example 5-23 Using the `ippsSum` Function

```
void sum(void) {  
    Ipp16s x[4] = {-32768, 32767, 32767, 32767}, sm;  
    ippsSum_16s_Sfs(x, 4, &sm, 1);  
    printf_16s("sum =", &sm, 1, ippStsNoErr);  
}
```

Output:

```
sum = 32766
```

Matlab* Analog:

```
>> x = [-32768, 32767, 32767, 32767]; sum(x)/2
```

Max

Returns the maximum value of a vector.

Syntax

```
IppStatus ippsMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax);  
IppStatus ippsMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax);  
IppStatus ippsMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax);  
IppStatus ippsMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax);
```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pMax</code>	Pointer to the output result.

len Number of elements in the vector

Description

The function `ippsMax` is declared in the `ipps.h` file. This function returns the maximum value of the input vector *pSrc*, and stores the result in *pMax*.

[Example 5-24](#) shows how to use the function `ippsMax_32f`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMax</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

MaxIndx

Returns the maximum value of a vector and the index of the maximum element.

Syntax

```
IppStatus ippsMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax, int* pIndx);
IppStatus ippsMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax, int* pIndx);
IppStatus ippsMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax, int* pIndx);
IppStatus ippsMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax, int* pIndx);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMax</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>pIndx</i>	Pointer to the index value of the maximum element.

Description

The function `ippsMaxIndx` is declared in the `ipps.h` file. This function returns the maximum value of the input vector `pSrc`, and stores the result in `pMax`. If `pIndx` is not a `NULL` pointer, the function returns the index of the maximum element and stores it in `pIndx`. If there are several equal maximum elements, the first index from the beginning is returned.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

MaxAbs

Returns the maximum absolute value of a vector.

Syntax

```
IppStatus ippsMaxAbs_16s(const Ipp16s* pSrc, int len, Ipp16s* pMaxAbs);  
IppStatus ippsMaxAbs_32s(const Ipp32s* pSrc, int len, Ipp32s* pMaxAbs);
```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pMaxAbs</code>	Pointer to the output result.
<code>len</code>	Number of elements in the vector.

Description

The function `ippsMaxAbs` is declared in the `ipps.h` file. This function returns the maximum absolute value of the input vector `pSrc`, and stores the result in `pMaxAbs`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMaxAbs</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

MaxAbsIdx

Returns the maximum absolute value of a vector and the index of the corresponding element.

Syntax

```
IppStatus ippsMaxAbsIdx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMaxAbs, int* pIdx);
IppStatus ippsMaxAbsIdx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMaxAbs, int* pIdx);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMaxAbs</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>pIdx</i>	Pointer to the index value of the maximum element.

Description

The function `ippsMaxAbsIdx` is declared in the `ipps.h` file. This function returns the maximum absolute value *pMaxAbs* of the input vector *pSrc*, and the index of the corresponding element *pIdx*. If there are several elements with the equal maximum absolute value, the first index from the beginning is returned.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Min

Returns the minimum value of a vector.

Syntax

```
IppStatus ippsMin_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin);
```

```
IppStatus ippMin_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin);  
IppStatus ippMin_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin);  
IppStatus ippMin_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMin</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.

Description

The function `ippMin` is declared in the `ipp.h` file. This function returns the minimum value of the input vector *pSrc*, and stores the result in *pMin*.

[Example 5-24](#) shows how to use the function `ippMin_32f`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMin</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Example 5-24 Using the `ippsMin` and `ippsMax` Functions

```
void minmax(void) {
    Ipp32f *x = ippsMalloc_32f(1000), minmax[2];
    int i;
    for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
    ippsMin_32f(x, 1000, &minmax[0]);
    ippsMax_32f(x, 1000, &minmax[1]);
    printf_32f("min max =", minmax, 2, ippsStsNoErr);
    ippsFree(x);
}
```

Output:

```
min max = 0.000855 0.999695
```

Matlab* Analog:

```
>> x = rand(1,1000); min(x), max(x)
```

MinIndx

Returns the minimum value of a vector and the index of the minimum element.

Syntax

```
IppStatus ippsMinIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, int* pIndx);
IppStatus ippsMinIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, int* pIndx);
IppStatus ippsMinIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, int* pIndx);
IppStatus ippsMinIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, int* pIndx);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMin</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.

pIndx Pointer to the index value of the minimum element.

Description

The function `ippsMinIndx` is declared in the `ipps.h` file. This function returns the minimum value of the input vector *pSrc* and stores the result in *pMin*. If *pIndx* is not a `NULL` pointer, the function returns the index of the minimum element and stores it in *pIndx*. If there are several equal minimum elements, the first index from the beginning is returned.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMin</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

MinAbs

Returns the minimum absolute value of a vector.

Syntax

```
IpplStatus ippsMinAbs_16s(const Ipp16s* pSrc, int len, Ipp16s* pMinAbs);  
IpplStatus ippsMinAbs_32s(const Ipp32s* pSrc, int len, Ipp32s* pMinAbs);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMinAbs</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsMinAbs` is declared in the `ipps.h` file. This function returns the minimum absolute value of the input vector *pSrc*, and stores the result in *pMinAbs*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMinAbs</i> or <i>pSrc</i> pointer is <code>NULL</code> .

`ippStsSizeErr` Indicates an error when *len* is less than or equal to 0.

MinAbsIndx

Returns the minimum absolute value of a vector and the index of the corresponding element.

Syntax

```
IppStatus ippsMinAbsIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMinAbs, int* pIndx);
IppStatus ippsMinAbsIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMinAbs, int* pIndx);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMinAbs</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>pIndx</i>	Pointer to the index value of the corresponding element.

Description

The function `ippsMinAbsIndx` is declared in the `ipps.h` file. This function returns the minimum absolute value *pMinAbs* of the input vector *pSrc*, and the index of the corresponding element *pIndx*. If there are several elements with equal maximum absolute value, the first index from the beginning is returned.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

MinMax

Returns the maximum and minimum values of a vector.

Syntax

```
IppStatus ippsMinMax_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin, Ipp8u* pMax);
IppStatus ippsMinMax_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin, Ipp16u* pMax);
IppStatus ippsMinMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, Ipp16s* pMax);
IppStatus ippsMinMax_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin, Ipp32u* pMax);
IppStatus ippsMinMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, Ipp32s* pMax);
IppStatus ippsMinMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, Ipp32f* pMax);
IppStatus ippsMinMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, Ipp64f* pMax);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMin</i>	Pointer to the minimum value.
<i>pMax</i>	Pointer to the maximum value.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsMinMax` is declared in the `ipps.h` file. This function returns the minimum and maximum values of the input vector *pSrc*, and stores the results in *pMin* and *pMax*, respectively.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMin</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

MinMaxIndx

Returns the maximum and minimum values of a vector and the indexes of the corresponding elements.

Syntax

```

IppStatus ippsMinMaxIndx_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin, int*
    pMinIndx, Ipp8u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin, int*
    pMinIndx, Ipp16u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, int*
    pMinIndx, Ipp16s* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin, int*
    pMinIndx, Ipp32u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, int*
    pMinIndx, Ipp32s* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, int*
    pMinIndx, Ipp32f* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, int*
    pMinIndx, Ipp64f* pMax, int* pMaxIndx);
    
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMin</i>	Pointer to the minimum value.
<i>pMax</i>	Pointer to the maximum value.
<i>len</i>	Number of elements in the vector.
<i>pMinIndx</i>	Pointer to the index value of the minimum element.
<i>pMaxIndx</i>	Pointer to the index value of the maximum element.

Description

The function `ippsMinMaxIndx` is declared in the `ipps.h` file. This function returns the minimum and maximum values of the input vector `pSrc` and stores the result in `pMin` and `pMax`, respectively. The function also returns the indexes of the minimum and maximum elements and stores them in `pMinIndx` and `pMaxIndx`, respectively. If there are several equal minimum or maximum elements, the first index from the beginning is returned.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Mean

Computes the mean value of a vector.

Syntax

```
IppStatus ippsMean_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean,
    IppHintAlgorithm hint);
IppStatus ippsMean_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pMean,
    IppHintAlgorithm hint);
IppStatus ippsMean_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean);
IppStatus ippsMean_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pMean);
IppStatus ippsMean_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean, int
    scaleFactor);
IppStatus ippsMean_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pMean, int
    scaleFactor);
```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pMean</code>	Pointer to the output result.
<code>len</code>	Number of elements in the vector

<i>hint</i>	Suggests using specific code. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments ”.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMean` is declared in the `ipps.h` file. This function computes the mean (average) of the vector *pSrc*, and stores the result in *pMean*. The mean of *pSrc* is defined by the formula:

$$mean = \frac{1}{len} \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

[Example 5-25](#) shows how to use the function `ippsMean_32f`

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMean</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Example 5-25 Using the `ippsMean` Function

```
void mean(void) {  
    Ipp32f *x = ippsMalloc_32f(1000), mean;  
    int i;  
    for(i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;  
    ippsMean_32f(x, 1000, &mean, ippAlgHintFast);  
    printf_32f("mean =", &mean, 1, ippStsNoErr);  
    ippsFree(x);  
}
```

Output:

mean = 0.492591

Matlab* Analog:

```
>> x = rand(1,1000); mean(x)
```

StdDev

Computes the standard deviation value of a vector.

Syntax

```
IppStatus ippsStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pStdDev,  
    IppHintAlgorithm hint);  
IppStatus ippsStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pStdDev);  
IppStatus ippsStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pStdDev,  
    int scaleFactor);  
IppStatus ippsStdDev_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pStdDev, int  
    scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pStdDev</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector

<i>hint</i>	Suggests using specific code. The values for the <i>hint</i> argument are described in Flag and Hint Arguments .
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsStdDev` is declared in the `ipps.h` file. This function computes the standard deviation of the input vector *pSrc*, and stores the result in *pStdDev*. The vector length can not be less than 2. The standard deviation of *pSrc* is defined by the unbiased estimate formula:

$$stdev = \sqrt{\frac{\sum_{n=0}^{len-1} (pSrc[n] - mean(pSrc))^2}{len - 1}}$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

[Example 5-26](#) shows how to use the function `ippsStdDev_32f`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pStdDev</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 1.

Example 5-26 Using the `ippsStdDev` Function

```
void stdev(void) {
    Ipp32f *x = ippsMalloc_32f(1000), stdev;
    int i;
    for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
    ippsStdDev_32f(x, 1000, &stdev, ippAlgHintFast);
    printf_32f("stdev =", &stdev, 1, ippStsNoErr);
    ippsFree(x);
}
```

Output:

```
stdev = 0.286813
```

Matlab* Analog:

```
>> x = rand(1,1000); std(x)
```

Norm

Computes the C, L1, L2, or L2Sqr norm of a vector.

Syntax

```
IppStatus ippsNorm_Inf_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_32fc32f(const Ipp32fc* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm,
    int scaleFactor);
```

```
IppStatus ippsNorm_L1_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L1_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_32fc64f(const Ipp32fc* pSrc, int len, Ipp64f* pNorm);
```

```

IppStatus ippsNorm_L1_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L1_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm,
    int scaleFactor);
IppStatus ippsNorm_L1_16s64s_Sfs(const Ipp16s* pSrc, int len, Ipp64s* pNorm, int
    scaleFactor);

IppStatus ippsNorm_L2_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_32fc64f(const Ipp32fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm, int
    scaleFactor);

IppStatus ippsNorm_L2Sqr_16s64s_Sfs(const Ipp16s* pSrc, int len, Ipp64s* pNorm,
    int scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the source vector <i>pSrc</i> .
<i>pNorm</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsNorm` is declared in the `ipps.h` file. This function computes the C, L1, L2, or L2Sqr norm of the source vector *pSrc* and stores the result in *pNorm*.

ippsNorm_Inf. The function `ippsNorm_Inf` computes the C norm defined by the formula:

$$Norm_C = \max_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L1. The function `ippsNorm_L1` computes the L1 norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L2. The function `ippsNorm_L2` computes the L2 norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc[n]|^2}$$

ippsNorm_L2Sqr. The function `ippsNorm_L2Sqr` computes the L2Sqr norm defined as square of the L2 norm.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the `scaleFactor` value.

Return Values

<code>ippsNoErr</code>	Indicates no error.
<code>ippsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pNorm</code> pointer is NULL.
<code>ippsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

NormDiff

Computes the C, L1, L2, or L2Sqr norm of two vectors' difference.

Syntax

```

IppStatus ippsNormDiff_Inf_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
    Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_32fc32f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp32s* pNorm, int scaleFactor);

```

```

IppStatus ippsNormDiff_L1_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
    Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_32fc64f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNormDiff_L1_16s64s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp64s* pNorm, int scaleFactor);

IppStatus ippsNormDiff_L2_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
    Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_32fc64f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp32s* pNorm, int scaleFactor);

IppStatus ippsNormDiff_L2Sqr_16s64s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp64s* pNorm, int scaleFactor);

```

Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the two source vectors; <i>pSrc2</i> can be NULL.
<i>pNorm</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsNorm` is declared in the `ipps.h` file. This function computes the C, L1, L2, or L2Sqr norm of the source vectors' difference, and stores the result in `pNorm`.

ippsNormDiff_Inf. The function `ippsNormDiff_Inf` computes the C norm defined by the formula:

$$Norm_{Inf} = \max_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L1. The function `ippsNormDiff_L1` computes the L1 norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L2. The function `ippsNormDiff_L2` computes the L2 norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|^2}$$

ippsNormDiff_L2Sqr. The function `ippsNormDiff_L2Sqr` computes the L2Sqr norm defined as square of the L2 norm.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the `scaleFactor` value.

[Example 5-27](#) shows how to use the function `ippsNormDiff`.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <code>pSrc1</code> , <code>pSrc2</code> , or <code>pNorm</code> pointer is <code>NULL</code> .
<code>ippsStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Example 5-27 Using the ippsNorm Function

```
int norm( void ) {
    Ipp16s x[LEN];
    Ipp32f Norm[3];
    IppStatus st;
    int i;
    for( i=0; i<LEN; ++i ) x[i] = (Ipp16s)rand();
    ippsNormDiff_Inf_16s32f( x, 0, LEN, Norm );
    ippsNormDiff_L1_16s32f( x, 0, LEN, Norm+1 );
    st = ippsNormDiff_L2_16s32f( x, 0, LEN, Norm+2 );
    printf_32f( "Norm (oo,L1,L2) =", Norm, 3, st );
    return Norm[2] <= Norm[1] && Norm[1] <= LEN*Norm[0];
}
```

Output:

```
Norm (oo,L1,L2) = 31993.000000 1526460.000000 180270.781250
```

Matlab* analog:

```
>> x = 32767*rand(1,100);norm(x,inf),norm(x,1),norm(x,2)
```

DotProd

Computes the dot product of two vectors.

Syntax

```
IppStatus ippsDotProd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp32f* pDp);
IppStatus ippsDotProd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int len,
    Ipp32fc* pDp);
IppStatus ippsDotProd_32f32fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp32fc* pDp);
IppStatus ippsDotProd_32f64f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp64f* pDp);
```

```
IppStatus ippsDotProd_32fc64fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp64fc* pDp);
IppStatus ippsDotProd_32f32fc64fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2,
    int len, Ipp64fc* pDp);
IppStatus ippsDotProd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
    Ipp64f* pDp);
IppStatus ippsDotProd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int len,
    Ipp64fc* pDp);
IppStatus ippsDotProd_64f64fc(const Ipp64f* pSrc1, const Ipp64fc* pSrc2, int
    len, Ipp64fc* pDp);
IppStatus ippsDotProd_16s64s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
    Ipp64s* pDp);
IppStatus ippsDotProd_16sc64sc(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, int
    len, Ipp64sc* pDp);
IppStatus ippsDotProd_16s16sc64sc(const Ipp16s* pSrc1, const Ipp16sc* pSrc2,
    int len, Ipp64sc* pDp);
IppStatus ippsDotProd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
    Ipp32f* pDp);
IppStatus ippsDotProd_16sc32fc(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, int
    len, Ipp32fc* pDp);
IppStatus ippsDotProd_16s16sc32fc(const Ipp16s* pSrc1, const Ipp16sc* pSrc2,
    int len, Ipp32fc* pDp);
IppStatus ippsDotProd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp16s* pDp, int scaleFactor);
IppStatus ippsDotProd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, int
    len, Ipp16sc* pDp, int scaleFactor);
IppStatus ippsDotProd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, int
    len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, int
    len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_16s16sc32sc_Sfs(const Ipp16s* pSrc1, const Ipp16sc*
    pSrc2, int len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16s32s32s_Sfs(const Ipp16s* pSrc1, const Ipp32s* pSrc2,
    int len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_16s16sc_Sfs(const Ipp16s* pSrc1, const Ipp16sc* pSrc2,
    int len, Ipp16sc* pDp, int scaleFactor);
```

```
IppStatus ippsDotProd_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_32s32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
    int len, Ipp32s* pDp, int scaleFactor);
```

Parameters

<i>pSrc1</i>	Pointer to the first vector to compute the dot product value.
<i>pSrc2</i>	Pointer to the second vector to compute the dot product value.
<i>pDp</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDotProd` is declared in the `ipps.h` file. This function computes the dot product (scalar value) of two vectors, *pSrc1* and *pSrc2*, and stores the result in *pDp*.

The computation is performed as follows:

$$dp = \sum_{n=0}^{len-1} pSrc1[n] \cdot pSrc2[n]$$

To compute the dot product of complex data, use the function `ippsConj` to conjugate one of the operands. The vectors *pSrc1* and *pSrc2* must be of equal length.

[Example 5-28](#) shows how to use the function `ippsDotProd_64f` to verify orthogonality of the sine and cosine functions. Two vectors are orthogonal to each other when the dot product of the two vectors is zero.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDp</i> , <i>pSrc1</i> , or <i>pSrc2</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Example 5-28 Using ippsDotProd to Verify Orthogonality of Sin and Cos

```

void dotprod(void) {
    Ipp64f x[10], dp;
    int n;
    for (n = 0; n<10; ++n) x[n] = sin(IPP_2PI * n / 8);
    ippsDotProd_64f(x, x+2, 8, &dp);
    printf_64f("dp =", &dp, 1, ippsNoErr);
}

```

Output:

```
dp = 0.000000
```

Matlab* Analog:

```
>> n = 0:9; x = sin(2*pi*n/8); a = x(1:8); b = x(3:10); a*b'
```

MaxEvery, MinEvery

Computes maximum or minimum value for each pair of elements of two vectors.

Syntax

```

IppStatus ippsMaxEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMaxEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMaxEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMinEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMinEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMinEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the first input vector.
<i>pSrcDst</i>	Pointer to the second input vector which stores the result.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsMaxEvery` is declared in the `ipps.h` file. This function computes the maximum between each pair of corresponding elements of two input vectors and stores the result in `pSrcDst`.

The function `ippsMinEvery` is declared in the `ipps.h` file. This function computes minimum values likewise.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Sampling Functions

The functions described in this section manipulate signal samples. Sampling functions are used to change the sampling rate of the input signal and thus to obtain the signal vector of a required length. The functions perform the following operations:

- Insert zero-valued samples between neighboring samples of a signal (up-sample).
- Remove samples from between neighboring samples of a signal (down-sample).

The upsampling and downsampling functions are used by some filtering functions described in Chapter 6.

SampleUp

Up-samples a signal, conceptually increasing its sampling rate by an integer factor.

Syntax

```
IppStatus ippsSampleUp_16s (const Ipp16s* pSrc, int srcLen, Ipp16s* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_32f (const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int*
    pDstLen, int factor, int* pPhase);
```

```

IppStatus ippsSampleUp_64f (const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_16sc (const Ipp16sc* pSrc, int srcLen, Ipp16sc* pDst,
    int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_32fc (const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst,
    int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_64fc (const Ipp64fc* pSrc, int srcLen, Ipp64fc* pDst,
    int* pDstLen, int factor, int* pPhase);

```

Parameters

<i>pSrc</i>	Pointer to the input array (the signal to be up-sampled).
<i>srcLen</i>	Number of samples in the input array <i>pSrc</i> .
<i>pDst</i>	Pointer to the output array.
<i>pDstLen</i>	Pointer to the length value of the output array <i>pDst</i> .
<i>factor</i>	Factor by which the signal is up-sampled. That is, <i>factor</i> - 1 zeros are inserted after each sample of the input array <i>pSrc</i> .
<i>pPhase</i>	Pointer to the input phase value which determines where each sample from <i>pSrc</i> lies within each output block of <i>factor</i> samples in <i>pDst</i> . The value of <i>pPhase</i> is required to be in the range [0; <i>factor</i> -1]. The output value of <i>pPhase</i> can be used for the next up-sampling with the same <i>factor</i> and next <i>pSrc</i> .

Description

The function `ippsSampleUp` is declared in the `ipps.h` file. This function up-samples the *srcLen*-length input array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Up-sampling inserts *factor*-1 zeros between each sample of *pSrc*. The *pPhase* argument determines where each sample from the input array lies within each output block of *factor* samples. The value of *pPhase* is required to be in the range [0; *factor*-1].

For example, if the input phase is 0, then every *factor* samples of the output array begin with the corresponding input array sample, the other *factor*-1 samples are equal to 0. The output array length is stored by the *pDstLen* address.

The *pPhase* value is the phase of an input array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

The `ippsSampleUp` functionality can be described as follows:

$$pDst[factor * n + phase] = pSrc[n], 0 \leq n < srcLen$$

$$pDst[factor * n + m] = 0, 0 \leq n < srcLen, 0 \leq m < factor, m \neq phase$$

$$pDstLen = factor * srcLen$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> , <code>pSrc</code> , <code>pDstLen</code> , or <code>pPhase</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>srcLen</code> is less than or equal to 0.
<code>ippStsSampleFactorErr</code>	Indicates an error when <code>factor</code> is less than or equal to 0.
<code>ippStsSamplePhaseErr</code>	Indicates an error when <code>pPhase</code> is negative, or bigger than or equal to <code>factor</code> .

SampleDown

Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.

Syntax

```

IppStatus ippsSampleDown_16s(const Ipp16s* pSrc, int srcLen, Ipp16s* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_16sc(const Ipp16sc* pSrc, int srcLen, Ipp16sc* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc* Dst, int*
    pDstLen, int factor, int* pPhase);

```


Parameters

<i>pSrc</i>	Pointer to the input array holding the samples to be down-sampled.
<i>srcLen</i>	Number of samples in the input array <i>pSrc</i> .
<i>pDst</i>	Pointer to the array that holds the output of the function <code>ippsSampleDown</code> .
<i>pDstLen</i>	Pointer to the length of the output array <i>pDst</i> .
<i>factor</i>	Factor by which the signal is down-sampled. That is, <i>factor</i> - 1 samples are discarded from every block of <i>factor</i> samples in <i>pSrc</i> .
<i>pPhase</i>	Pointer to the input phase value that determines which of the samples within each block of <i>factor</i> samples from <i>pSrc</i> is not discarded and copied to <i>pDst</i> . The value of <i>pPhase</i> is required to be in the range [0; <i>factor</i> -1]. The output value of <i>pPhase</i> can be used for the next down-sampling (sub-sampling) with the same <i>factor</i> and next <i>pSrc</i> .

Description

The function `ippsSampleDown` is declared in the `ipps.h` file. This function down-samples the *srcLen*-length array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Down-sampling discards *factor* - 1 samples from *pSrc*, copying one sample from each block of *factor* samples from *pSrc* to *pDst*. The *pPhase* argument determines which of the samples in each block is not discarded and where it lies within each input block of *factor* samples. The value of *pPhase* is required to be in the range [0; *factor*-1]. The output array length is stored by the *pDstLen* address.

The *pPhase* value is the phase of an input array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

You can use the FIR multi-rate filter to combine filtering and resampling, for example, for antialiasing filtering before the sub-sampling procedure.

The `ippsSampleDown` functionality can be described as follows:

```


pDstLen = (srcLen + factor - 1 - phase)/factor



pDst[n] = pSrc[factor * n + phase], 0 ≤ n < pDstLen



phase = (factor + phase - srcLen%factor)%factor


```

[Example 5-29](#) shows how to use the function `ippsSampleDown`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> , <i>pSrc</i> , <i>pDstLen</i> , or <i>pPhase</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>srcLen</i> is less than or equal to 0.
<code>ippStsSampleFactorErr</code>	Indicates an error when <i>factor</i> is less than or equal to 0.
<code>ippStsSamplePhaseErr</code>	Indicates an error when <i>pPhase</i> is negative, or bigger than or equal to <i>factor</i> .

Example 5-29 Using the `ippsSampleDown` Function

```
void sampling( void ) {
    Ipp16s x[8] = { 1,2,3,4,5,6,7,8 };
    Ipp16s y[8] = { 9,10,11,12,13,14,15,16 }, z[8];
    int dstLen1, dstLen2, phase = 2;
    IppStatus st = ippsSampleDown_16s(x, 8, z, &dstLen1, 3, &phase);
    st = ippsSampleDown_16s(y, 8, z+dstLen1, &dstLen2, 3, &phase);
    printf_16s("down-sampling =", z, dstLen1+dstLen2, st);
}
```

Output:

```
down-sampling = 3 6 9 12 15
```

Filtering Functions

6

This chapter describes Intel[®] IPP functions that perform convolution and correlation operations, as well as linear and non-linear filtering. These functions are grouped into the following sections:

[Convolution and Correlation Functions](#)

[Filtering Functions](#)

Each section starts with a table that lists functions described in more detail later in this section, together with the brief description of operations that these functions perform.

Convolution and Correlation Functions

Convolution is an operation used to define an output signal from any linear time-invariant (LTI) processor in response to any input signal.

The correlation functions described later in this section estimate either the auto-correlation of a source vector or the cross-correlation of two vectors

The full list of functions in this group is given in [Table 6-1](#).

Table 6-1 Intel IPP Convolution and Correlation Functions

Function Base Name	Operation
Conv	Performs finite, linear convolution of two sequences.
ConvBiased	Computes the specified number of elements of the full finite linear convolution of two vectors.
ConvCyclic	Performs cyclic convolution of two sequences of the fixed size.
AutoCorr	Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.

Table 6-1 Intel IPP Convolution and Correlation Functions (continued)

Function Base Name	Operation
CrossCorr	Estimates the cross-correlation of two vectors.
UpdateLinear	Integrates an input vector with specified integration weight.
UpdatePower	Integrates the square of an input vector with specified integration weight

Conv

Performs finite, linear convolution of two vectors.

Syntax

```
IppStatus ippsConv_32f(const Ipp32f* pSrc1, int src1Len, const Ipp32f* pSrc2,
    int src2Len, Ipp32f* pDst);
IppStatus ippsConv_64f(const Ipp64f* pSrc1, int src1Len, const Ipp64f* pSrc2,
    int src2Len, Ipp64f* pDst);
IppStatus ippsConv_16s_Sfs(const Ipp16s* pSrc1, int src1Len, const Ipp16s*
    pSrc2, int src2Len, Ipp16s* pDst, int scaleFactor);
```

Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the source vectors.
<i>src1Len</i>	Number of elements in the vector <i>pSrc1</i> .
<i>src2Len</i>	Number of elements in the vector <i>pSrc2</i> .
<i>pDst</i>	Pointer to the destination vector.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsConv` is declared in the `ipps.h` file. This function performs finite linear convolution of two sequences. The *src1Len*-length vector *pSrc1* is convolved with the *src2Len*-length vector *pSrc2* to produce an $(src1Len + src2Len - 1)$ -length vector *pDst*. The result of the convolution is defined as follows:

$$pDst[n] = \sum_{k=0}^n pSrc1[k] \cdot pSrc2[n-k] \quad 0 \leq n \leq src1Len + src2Len - 1$$

Here $pSrc1[i] = 0$, if $i \geq src1Len$, and $pSrc2[j] = 0$, if $j \geq src2Len$.

[Example 6-1](#) shows the code for the convolution of two vectors using `ippsConv_16s_Sfs` function.

Example 6-1 Using the `ippsConv` Function to Convolve Two Vectors

```
IppStatus convolution(void) {
    Ipp16s x[5] = {-2,0,1,-1,3}, h[2] = {0,1}, y[6];
    IppStatus st = ippsConv_16s_Sfs(x, 5, h, 2, y, 0);
    printf_16s("conv =", y, 6, st);
    return st;
}
```

Output:

```
conv = 0 -2 0 1 -1 3
```

Matlab* Analog:

```
>> x = [-2,0,1,-1,3]; h = [0,1]; y = conv(x,h)
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>src1Len</i> or <i>src2Len</i> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for internal buffers.

ConvBiased

Computes the specified number of elements of the full finite linear convolution of two vectors.

Syntax

```
IppStatus ippsConvBiased_32f(const Ipp32f* pSrc1, int src1Len, const Ipp32f* pSrc2,
    int src2Len, Ipp32f* pDst, int dstLen, int bias);
```

Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the two vectors to be convolved.
<i>src1Len</i>	Number of elements in the vector <i>pSrc1</i> .
<i>src2Len</i>	Number of elements in the vector <i>pSrc2</i> .
<i>pDst</i>	Pointer to the vector <i>pDst</i> . This vector stores the result of the convolution.
<i>dstLen</i>	Number of elements in the vector <i>pDst</i> .
<i>bias</i>	Parameter that specifies the starting element of the convolution.

Description

The function `ippsConvBiased` is declared in the `ipps.h` file. This function computes *dstLen* elements of finite linear convolution of two specified vectors *pSrc1* and *pSrc2* starting with an element that is specified by the *bias*. The result is stored in the vector *pDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>src1Len</i> or <i>src2Len</i> is less than or equal to 0.

ConvCyclic

Performs cyclic convolution of two sequences of the fixed size.

Syntax

```
IppStatus ippsConvCyclic8x8_32f(const Ipp32f* x, const Ipp32f* h, Ipp32f* y);  
IppStatus ippsConvCyclic4x4_32f32fc(const Ipp32f* x, const Ipp32fc* h, Ipp32fc* y);  
IppStatus ippsConvCyclic8x8_16s_Sfs(const Ipp16s* x, const Ipp16s* h, Ipp16s* y,  
    int scaleFactor);
```

Parameters

<i>x, h</i>	Pointers to the source vectors.
<i>y</i>	Pointer to the destination vector that stores the result of convolution.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Discussion

The function `ippsConvCyclic` is declared in the `ipps.h` file. This function performs cyclic convolution of two sequences of the fixed size.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

AutoCorr

Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.

Syntax

```
IppStatus ippsAutoCorr_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int dstLen);  
IppStatus ippsAutoCorr_NormA_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst,  
    int dstLen);
```

```

IppStatus ippsAutoCorr_NormB_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst,
    int dstLen);
IppStatus ippsAutoCorr_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int
    dstLen);
IppStatus ippsAutoCorr_NormA_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst,
    int dstLen);
IppStatus ippsAutoCorr_NormB_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst,
    int dstLen );
IppStatus ippsAutoCorr_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst, int
    dstLen);
IppStatus ippsAutoCorr_NormA_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc*
    pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc*
    pDst, int dstLen);
IppStatus ippsAutoCorr_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc* pDst, int
    dstLen);
IppStatus ippsAutoCorr_NormA_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc*
    pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc*
    pDst, int dstLen);
IppStatus ippsAutoCorr_16s_Sfs(const Ipp16s* pSrc, int srcLen, Ipp16s* pDst,
    int dstLen, int scaleFactor );
IppStatus ippsAutoCorr_NormA_16s_Sfs( const Ipp16s* pSrc, int srcLen, Ipp16s*
    pDst, int dstLen, int scaleFactor );
IppStatus ippsAutoCorr_NormB_16s_Sfs(const Ipp16s* pSrc, int srcLen, Ipp16s*
    pDst, int dstLen, int scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>srcLen</i>	The number of elements in the source vector.
<i>pDst</i>	Pointer to the destination vector, which stores the estimated auto-correlation results of the source vector.
<i>dstLen</i>	The number of elements in the destination vector (length of auto-correlation).
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The `ippsAutoCorr` function is declared in the `ipps.h` file. This function estimates normal auto-correlation of the `srcLen`-length source vector `pSrc` and stores the results in the `dstLen`-length vector `pDst`. Function flavors `ippsAutoCorr_NormA` and `ippsAutoCorr_NormB` compute biased and unbiased auto-correlation of the source vector, respectively. The resulting vector `pDst` is defined by the following equations:

$$pDst[n] = \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \leq n < dstLen \quad (\text{normal})$$

$$pDst[n] = \frac{1}{srcLen} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \leq n < dstLen \quad (\text{biased})$$

$$pDst[n] = \frac{1}{srcLen-n} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \leq n < dstLen \quad (\text{unbiased})$$

where

$$pSrc[i] = \begin{cases} pSrc[i], & 0 \leq i < srcLen \\ 0, & \text{otherwise} \end{cases}$$

Application Note: The auto-correlation estimates are computed only for positive lags, since the auto-correlation for a negative lag value is the complex conjugate of the auto-correlation for the equivalent positive lag.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>srcLen</code> or <code>dstLen</code> is less than or equal to 0.

See Also

[CrossCorr](#)

Estimates the cross-correlation of two vectors.

CrossCorr

*Estimates the cross-correlation
of two vectors.*

Syntax

```
IppStatus ippsCrossCorr_32f(const Ipp32f* pSrc1, int len1, const Ipp32f* pSrc2,
    int len2, Ipp32f* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64f(const Ipp64f* pSrc1, int len1, const Ipp64f* pSrc2,
    int len2, Ipp64f* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_32fc(const Ipp32fc* pSrc1, int len1, const Ipp32fc* pSrc2,
    int len2, Ipp32fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64fc(const Ipp64fc* pSrc1, int len1, const Ipp64fc* pSrc2,
    int len2, Ipp64fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_16s_Sfs(const Ipp16s* pSrc1, int len1, const Ipp16s* pSrc2,
    int len2, Ipp16s* pDst, int dstLen, int lowLag, int scaleFactor);
```

Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>len1</i>	Number of elements in the vector <i>pSrc1</i> .
<i>pSrc2</i>	Pointer to the second source vector.
<i>len2</i>	Number of elements in the vector <i>pSrc2</i> .
<i>pDst</i>	Pointer to the vector which stores the results of the estimated cross-correlation of the vectors <i>pSrc1</i> and <i>pSrc2</i> .
<i>dstLen</i>	Number of elements in the vector <i>pDst</i> , which determines the range of lags at which the correlation estimates are computed.
<i>lowLag</i>	Lower value of the range of lags at which the correlation estimates are computed.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsCrossCorr` is declared in the `ipps.h` file. This function estimates cross-correlation of the `len1`-length vector `pSrc1` and the `len2`-length vector `pSrc2`, and stores the results in the vector `pDst`.

The resulting vector `pDst` is defined by the equation:

$$pDst[n] = \sum_{i=0}^{len1-1} conj(pSrc1[i]) \cdot pSrc2[n+i+lowLag] ,$$

where $0 \leq n < dstLen$, and

$$pSrc2[j] = \begin{cases} pSrc2[j], & 0 \leq j < len2 \\ 0, & otherwise \end{cases}$$

[Example 6-2](#) shows how to use the function `ippsCrossCorr`.

Example 6-2 Using the `ippsCrossCorr` Function

```
void crossCorr(void) {
    #undef LEN
    #define LEN 11
    Ipp32f win[LEN], y[LEN];
    IppStatus st;
    ippsSet_32f (1, win, LEN);
    ippsWinHamming_32f_I (win, LEN);
    st = ippsCrossCorr_32f (win, LEN, win, LEN, y, LEN, -(LEN-1));
    printf_32f("cross corr =", y, 7, st);
}
```

Output:

```
cross corr = 0.006400 0.026856 0.091831 0.242704 0.533230
1.009000 1.672774
```

Matlab* analog:

```
>> x = hamming(11)'; y = xcorr(x,x); y(1:7)
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc1</code> or <code>pSrc2</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len1</code> or <code>len2</code> is less than or equal to 0.

UpdateLinear

Integrates an input vector with specified integration weight.

Syntax

```
IppStatus ippsUpdateLinear_16s32s_I(const Ipp16s* pSrc, int len, Ipp32s* pSrcDst,
    int srcShiftRight, Ipp16s alpha, IppHintAlgorithm hint);
```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>len</code>	Number of elements of the vector.
<code>pSrcDst</code>	Pointer to the input value and output result.
<code>srcShiftRight</code>	Shift value; must be non-negative.
<code>alpha</code>	Integration weight.
<code>hint</code>	Suggests using specific code. The values for the <code>hint</code> argument are described in “ Flag and Hint Arguments .”

Description

The function `ippsUpdateLinear` is declared in the `ipps.h` file. This function performs `len` iterations in which the sum

$$\alpha * pSrcDst + (1 - \alpha) * pSrc[i]_{shift}$$

is calculated and stored in `pSrcDst`.

Here `i` is the number of previous iterations, `pSrcDst` is the result of previous iteration, and `pSrc[i]shift` is a source vector element right-shifted by the non-negative value `srcShiftRight`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

UpdatePower

Integrates the square of an input vector with specified integration weight.

Syntax

```
IppStatus ippUpdatePower_16s32s_I(const Ipp16s* pSrc, int len, Ipp32s* pSrcDst,  
    int srcShiftRight, Ipp16s alpha, IppHintAlgorithm hint);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>len</i>	Number of elements of the vector.
<i>pSrcDst</i>	Pointer to input and output
<i>srcShiftRight</i>	Shift value.
<i>alpha</i>	Integration weight.
<i>hint</i>	Suggests using specific code. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments .”

Description

The function `ippUpdatePower` is declared in the `ipps.h` file. This function performs *len* iterations in which the sum

$$\alpha * pSrcDst + (1 - \alpha) * pSrc[i]_{shift} * pSrc[i]_{shift}$$
is calculated and stored in *pSrcDst*.

Here *i* is the number of previous iterations, *pSrcDst* is the result of previous iteration, and *pSrc[i]_{shift}* is a source vector element right-shifted by the non-negative value *srcShiftRight*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Filtering Functions

The Intel IPP functions described in this section implement the following types of filters:

- finite impulse response (FIR) filter
- adaptive finite impulse response using least mean squares (LMS) filter
- infinite impulse response (IIR) filter
- median filter.

A special set of functions is designed to generate filter coefficients for different types of FIR filters.

The full list of filtering functions is given in [Table 6-2](#)

Table 6-2 Intel IPP Filtering Functions

Function Base Name	Operation
SumWindow	Initializes a single-rate FIR filter state structure.
FIR Filter Functions	
FIRInitAlloc	Allocates memory and initializes a single-rate FIR filter state structure.
FIRMRInitAlloc	Allocates memory and initializes a multi-rate FIR filter state structure.
FIRFree	Frees memory allocated for the FIR filter state structure.
FIRInit	Initializes a single-rate FIR filter state structure.
FIR_MRInit	Initializes a multi-rate FIR filter state structure.
FIRGetStateSize, FIRMRGetStateSize	Computes the size of an external buffer for FIR filter structure.
ippsFIRGetTaps	Retrieves the tap values from the FIR filter state structure.
ippsFIRSetTaps	Sets the taps values in the FIR filter state structure.
FIRGetDlyLine	Retrieves the delay line contents from the FIR filter state structure.
FIRSetDlyLine	Sets the delay line contents in the FIR filter state structure.
FIROne	Filters a single sample through a FIR filter.
FIR	Filters a block of samples through a FIR filter.
Direct versions	
FIROne_Direct	Directly filters a single sample through a FIR filter.

Table 6-2 Intel IPP Filtering Functions (continued)

Function Base Name	Operation
FIR_Direct	Directly filters a block of samples through a single-rate FIR filter.
FIRMR_Direct	Directly filters a block of samples through a multi-rate FIR filter.
Sparse FIR Filter	
FIRSparseInit	Initializes a sparse FIR filter structure.
FIRSparseGetStateSize	Computes the size of the external buffer for the sparse FIR filter structure.
FIRSparse	Filters a block of samples through a sparse FIR filter.
FIR Filter Coefficient Generating Functions	
FIRGenLowpass	Computes the lowpass FIR filter coefficients.
FIRGenHighpass	Computes the highpass FIR filter coefficients.
FIRGenBandpass	Computes the bandpass FIR filter coefficients.
FIRGenBandstop	Computes the bandstop FIR filter coefficients.
Single-Rate FIR LMS Filter Functions	
FIRLMSInitAlloc	Allocates memory and initializes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.
FIRLMSFree	Closes an adaptive FIR filter that uses the LMS algorithm.
FIRLMSGetTaps	Gets the taps of a FIR LMS filter.
FIRLMSGetDlyLine	Retrieves the delay line contents from the FIR LMS filter.
FIRLMSSetDlyLine	Sets the delay line contents in the FIR LMS filter.
FIRLMS	Filters an array through a FIR LMS filter.
Direct versions	
FIRLMSOne_Direct	Filters a single sample through a FIR LMS filter.
Multi-Rate FIR LMS Filter Functions	
FIRLMSMRInitAlloc	Allocates memory and initializes an adaptive multi-rate FIR filter that uses the least mean squares (LMS) algorithm.
FIRLMSMRFree	Closes an adaptive multi-rate FIR filter that uses the least mean squares algorithm.
FIRLMSMRSetMu	Sets the adaptation step.
FIRLMSMRUpdateTaps	Updates the filter coefficients using the adaptation error value.
FIRLMSMRGetTaps	Retrieves tap values in the multi-rate FIR LMS filter.
FIRLMSMRSetTaps	Sets tap values in the multi-rate FIR LMS filter.

Table 6-2 Intel IPP Filtering Functions (continued)

Function Base Name	Operation
<u>FIRLMSMRGetTapsPointer</u>	Returns the pointer to the filter coefficients.
<u>FIRLMSMRGetDlyLine</u>	Retrieves the delay line contents from the multi-rate FIR LMS filter state.
<u>FIRLMSMRSetDlyLine</u>	Sets the delay line contents in the multi-rate FIR LMS filter state.
<u>FIRLMSMRGetDlyVal</u>	Gets one delay line values from the specified position.
<u>FIRLMSMRPutVal</u>	Places the input value in the delay line.
<u>FIRLMSMROne</u>	Filters data placed in the delay line.
<u>FIRLMSMROneVal</u>	Filters one input value.
IIR Filter Functions	
<u>IIRInitAlloc</u>	Allocates memory and initializes an arbitrary IIR filter state.
<u>IIRInitAlloc_BiQuad</u>	Allocates memory and initializes a biquad IIR filter state.
<u>IIRFree</u>	Closes an IIR filter state.
<u>IIRInit</u>	Initializes an arbitrary IIR filter state.
<u>IIRInit_BiQuad</u>	Initializes a biquad IIR filter state
<u>IIRGetStateSize</u>	Computes the length of the external buffer for the arbitrary IIR filter state structure.
<u>IIRGetStateSize_BiQuad</u>	Computes the length of the external buffer for the biquad IIR filter state structure.
<u>IIRSetTaps</u>	Sets the taps in an IIR filter state
<u>IIRGetDlyLine</u>	Retrieves the delay line values from the IIR filter state.
<u>IIRSetDlyLine</u>	Sets the delay line values in the IIR filter state.
<u>IIROne</u>	Filters a single sample through an IIR filter.
<u>IIR</u>	Filters a block of samples through an IIR filter.
Direct versions	
<u>IIROne_Direct</u>	Directly filters a single sample through the arbitrary IIR filter.
<u>IIROne_BiQuadDirect</u>	Directly filters a single sample through the biquad IIR filter.
<u>IIR_Direct</u>	Directly filters a block of samples through the arbitrary IIR filter.
<u>IIR_BiQuadDirect</u>	Directly filters a block of samples through the biquad IIR filter
Sparse IIR Filter	
<u>IIRSparseInit</u>	Initializes a sparse IIR filter structure.

Table 6-2 Intel IPP Filtering Functions (continued)

Function Base Name	Operation
IIRSparseGetStateSize	Computes the size of the external buffer for the sparse IIR filter structure.
IIRSparse	Filters a block of samples through a sparse IIR filter.
Median Filter Functions	
FilterMedian	Computes median values for each input array element.

SumWindow

Sums elements in the mask applied to each element of a vector.

Syntax

```
IppStatus ippsSumWindow_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len, int maskSize);  
IppStatus ippsSumWindow_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len, int maskSize);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements of the vector.
<i>maskSize</i>	Size of the mask.

Description

The function `ippsSumWindow` is declared in the `ipps.h` file. This function sets each element in the destination vector *pDst* as the sum of *maskSize* elements of the source vector *pSrc*. The computation is performed as follows:

$$pDst[n] = \sum_{k=n}^{maskSize} pSrc[k] \quad , 0 \leq n < len$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsMaskSizeErr</code>	Indicates an error when <i>maskSize</i> is less than or equal to 0.

FIR Filter Functions

The functions described in this section perform a finite impulse response filtering of input data. The functions initialize a finite impulse response filter, get and set the delay lines and filter coefficients (taps), and perform filtering. To use the FIR filter functions, follow this general scheme:

1. Call either [ippSFIRInitAlloc](#) or [ippSFIMRInitAlloc](#) to allocate memory and initialize the taps and the delay line in the filter state structure of a single-rate or multi-rate filter, respectively.
Or call either [ippSFIRInit](#) or [ippSFIR_MRInit](#) to initialize the taps and the delay line in the corresponding filter state structure in the previously created external buffer. The size of this buffer should be computed beforehand by calling the functions [ippSFIRGetStateSize](#) or [ippSFIMRGetStateSize](#), respectively.
2. Call [ippSFIROne](#) to filter a single sample through a single-rate filter and/or call [ippSFIR](#) to filter a block of consecutive samples through a single-rate or multi-rate filter.
3. To set new taps and delay line values in the previously initialized filter state, call the functions [ippSFIRSetTaps](#) and [ippSFIRSetDlyLine](#). To get taps and delay line values of the initialized filter state, call the functions [ippSFIRGetTaps](#) and [ippSFIRSetDlyLine](#).
4. Call [ippSFIRFree](#) to free dynamic memory associated with the FIR filter state structure created by [ippSFIRInitAlloc](#) or [ippSFIMRInitAlloc](#).

Alternatively, you may use the direct version of the functions [ippSFIROne_Direct](#), [ippSFIR_Direct](#), [ippSFIMR_Direct](#). These functions perform filtering without initializing the filter state structure. All required parameters are directly set in the function.

Special [set of functions](#) allows to compute the filter coefficients for different filters.

FIRInitAlloc

Allocates memory and initializes a single-rate FIR filter state.

Syntax

Case 1: Operation on integer samples.

```

IppStatus ippsFIRInitAlloc32s_16s(IppsFIRState32s_16s** ppState, const Ipp32s*
    pTaps, int tapsLen, int tapsFactor, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32s_16s32f(IppsFIRState32s_16s** ppState, const
    Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32f_16s(IppsFIRState32f_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc64f_16s(IppsFIRState64f_16s** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc_32s(IppsFIRState_32s** ppState, const Ipp32s* pTaps,
    int tapsLen, const Ipp32s* pDlyLine);
IppStatus ippsFIRInitAlloc64f_32s(IppsFIRState64f_32s** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp32s* pDlyLine);

```

```

IppStatus ippsFIRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, const Ipp32sc* pDlyLine);

```

Case 2: Operation on floating point samples.

```

IppStatus ippsFIRInitAlloc_32f(IppsFIRState_32f** ppState, const Ipp32f* pTaps,
    int tapsLen, const Ipp32f* pDlyLine);
IppStatus ippsFIRInitAlloc64f_32f(IppsFIRState64f_32f** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp32f* pDlyLine);

```

```

IppStatus ippsFIRInitAlloc_64f(IppsFIRState_64f** ppState, const Ipp64f* pTaps,
    int tapsLen, const Ipp64f* pDlyLine);

IppStatus ippsFIRInitAlloc_32fc(IppsFIRState_32fc** ppState, const Ipp32fc* pTaps,
    int tapsLen, const Ipp32fc* pDlyLine);

IppStatus ippsFIRInitAlloc_64fc_32fc(IppsFIRState_64fc_32fc** ppState, const Ipp64fc*
    pTaps, int tapsLen, const Ipp32fc* pDlyLine);

IppStatus ippsFIRInitAlloc_64fc(IppsFIRState_64fc** ppState, const Ipp64fc* pTaps,
    int tapsLen, const Ipp64fc* pDlyLine);

```

Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of Ipp32s data type (for integer versions only).
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>tapsLen</i> .
<i>ppState</i>	Pointer to the pointer to the FIR state structure to be created.

Description

The function `ippsFIRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a single-rate FIR filter state. The initialization function copies the tap values from the *tapsLen*-length array *pTaps* into the state structure *ppState*. To scale integer taps use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the *tapsLen*-length array *pDlyLine* is not NULL, the array content is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

Note that the delay line length is different than that for direct FIR filters (where this length is doubled).

If the state is not created, the initialization function returns an error status.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRMRInitAlloc

Allocates memory and initializes a multi-rate FIR filter state.

Syntax

Case 1: Operation on integer samples.

```

IppStatus ippFIRMRInitAlloc32s_16s(IppsFIRState32s_16s** ppState, const
    Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16s* pDlyLine);

IppStatus ippFIRMRInitAlloc32s_16s32f(IppsFIRState32s_16s** ppState, const
    Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine);

IppStatus ippFIRMRInitAlloc32f_16s(IppsFIRState32f_16s** ppState, const
    Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine);

IppStatus ippFIRMRInitAlloc64f_16s(IppsFIRState64f_16s** ppState, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine);

IppStatus ippFIRMRInitAlloc64f_32s(IppsFIRState64f_32s** ppState, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32s* pDlyLine);

IppStatus ippFIRMRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16sc* pDlyLine);

IppStatus ippFIRMRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc** ppState,
    const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16sc* pDlyLine);

IppStatus ippFIRMRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine);

```

```
IppStatus ippsFIRMRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32sc* pDlyLine);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsFIRMRInitAlloc_32f(IppsFIRState_32f** ppState, const Ipp32f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc64f_32f(IppsFIRState64f_32f** ppState, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc_64f(IppsFIRState_64f** ppState, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp64f* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc_32fc(IppsFIRState_32fc** ppState, const Ipp32fc*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc_64fc(IppsFIRState_64fc** ppState, const Ipp64fc*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp64fc* pDlyLine);
```

Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of Ipp32s data type (for integer versions only).
<i>downFactor</i>	Downsampling factor.
<i>downPhase</i>	Phase for downsampled signal.
<i>upFactor</i>	Upsampling factor.

<i>upPhase</i>	Phase for upsampled signal.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array $(\text{tapsLen} + \text{upFactor} - 1) / \text{upFactor}$.
<i>ppState</i>	Pointer to the pointer to the FIR state structure to be created.

Description

The function `ippsFIRMRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a multi-rate FIR filter state. The initialization functions copy the taps from the *tapsLen*-length array *pTaps* into the state structure *pState*. To scale integer taps use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

If the state is not created, the initialization function returns an error status.

The function initializes the state structure *pState* of a multi-rate filter; that is, a filter that internally upsamples and/or downsamples using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the [ippsSampleUp](#) function for more details). That is, *upFactor*-1 zeros are inserted between each sample of input signal.

The argument *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see description of the [ippsSampleDown](#) function for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The argument *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response.

If the delay line array *pDlyLine* is non-NULL, its length is defined as $(\text{tapsLen} + \text{upFactor} - 1) / \text{upFactor}$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error when <i>upFactor</i> (<i>downFactor</i>) is less than or equal to 0.
<code>ippStsFIRMRPhaseErr</code>	Indicates an error when <i>upPhase</i> (<i>downPhase</i>) is negative, or greater than or equal to <i>upFactor</i> (<i>downFactor</i>).
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRFree

Closes a FIR filter state.

Syntax

```

IppStatus ippFIRFree32s(IppsFIRState32s_16s* pState);
IppStatus ippFIRFree32f_16s(IppsFIRState32f_16s* pState);
IppStatus ippFIRFree64f_16s(IppsFIRState64f_16s* pState);
IppStatus ippFIRFree_32s(IppsFIRState_32s* pState);
IppStatus ippFIRFree64f_32s(IppsFIRState64f_32s* pState);
IppStatus ippFIRFree32sc_16sc(IppsFIRState32sc_16sc* pState);
IppStatus ippFIRFree32fc_16sc(IppsFIRState32fc_16sc* pState);
IppStatus ippFIRFree64fc_16sc(IppsFIRState64fc_16sc* pState);
IppStatus ippFIRFree64fc_32sc(IppsFIRState64fc_32sc* pState);

IppStatus ippFIRFree_32f(IppsFIRState_32f* pState);
IppStatus ippFIRFree64f_32f(IppsFIRState64f_32f* pState);
IppStatus ippFIRFree_64f(IppsFIRState_64f* pState);
IppStatus ippFIRFree_32fc(IppsFIRState_32fc* pState);
IppStatus ippFIRFree64fc_32fc(IppsFIRState64fc_32fc* pState);
IppStatus ippFIRFree_64fc(IppsFIRState_64fc* pState);

```

Parameters

pState Pointer to the FIR filter state structure to be closed.

Description

The function `ippsFIRFree` is declared in the `ipps.h` file. This function closes the FIR filter state by freeing all memory associated with the filter state structure created by [ippsFIRInitAlloc](#) or [ippsFIRMRIInitAlloc](#). Call `ippsFIRFree` after filtering is completed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRInit

Initializes a single-rate FIR filter state.

Syntax

Case 1: Operation on integer samples.

```

IppStatus ippsFIRInit32s_16s(IppsFIRState32s_16s** ppState, const Ipp32s*
    pTaps, int tapsLen, int tapsFactor, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit32s_16s32f(IppsFIRState32s_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit32f_16s(IppsFIRState32f_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64f_16s(IppsFIRState64f_16s** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit_32s(IppsFIRState_32s** ppState, const Ipp32s* pTaps, int
    tapsLen, const Ipp32s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64f_32s(IppsFIRState64f_32s** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit32sc_16sc(IppsFIRState32sc_16sc** ppState, const Ipp32sc*
    pTaps, int tapsLen, int tapsFactor, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

```

```

IppStatus ippsFIRInit32sc_16sc32fc(IppsFIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit32fc_16sc(IppsFIRState32fc_16sc** ppState, const Ipp32fc*
    pTaps, int tapsLen, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64fc_16sc(IppsFIRState64fc_16sc** ppState, const Ipp64fc*
    pTaps, int tapsLen, const Ipp16sc* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRInit64fc_32sc(IppsFIRState64fc_32sc** ppState, const Ipp64fc*
    pTaps, int tapsLen, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);

```

Case 2: Operation on floating point samples.

```

IppStatus ippsFIRInit_32f(IppsFIRState_32f** ppState, const Ipp32f* pTaps, int
    tapsLen, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64f_32f(IppsFIRState64f_32f** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit_64f(IppsFIRState_64f** ppState, const Ipp64f* pTaps, int
    tapsLen, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit_32fc(IppsFIRState_32fc** ppState, const Ipp32fc* pTaps,
    int tapsLen, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64fc_32fc(IppsFIRState64fc_32fc** ppState, const Ipp64fc*
    pTaps, int tapsLen, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit_64fc(IppsFIRState_64fc** ppState, const Ipp64fc* pTaps,
    int tapsLen, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

```

Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of Ipp32s data type (for integer versions only).
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>tapsLen</i> .
<i>ppState</i>	Pointer to the pointer to the FIR state structure to be created.
<i>pBuffer</i>	Pointer to the external buffer for FIR state structure.

Description

The function `ippsFIRInit` is declared in the `ipps.h` file. This function initializes a single-rate FIR filter state structure in the external buffer. The size of this buffer should be computed previously by calling the functions [ippsFIRGetStateSize](#). The initialization function copies the taps from the `tapsLen` -length array `pTaps` into the state structure `ppState` of a single-rate filter. To scale integer taps, use the `tapsFactor` value.

The `tapsLen` -length array `pDlyLine` specifies the delay line values. If the pointer to the array `pDlyLine` is not `NULL`, the array contents is copied into the state structure `pState`, otherwise the delay line values in the state structure are initialized to 0.

Note that the delay line length is different than that for direct FIR filters (where this length is doubled).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFIRLenErr</code>	Indicates an error when <code>tapsLen</code> is less than or equal to 0.

FIR MRInit

Initializes a multi-rate FIR filter state.

Syntax

Case 1: Operation on integer samples.

```

IppStatus ippsFIRMRInit32s_16s(IppsFIRState32s_16s** ppState, const Ipp32s*
    pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32s_16s32f(IppsFIRState32s_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32f_16s(IppsFIRState32f_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64f_16s(IppsFIRState64f_16s** ppState, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);

```

```

IppStatus ippsFIRMRInit64f_32s(IppsFIRState64f_32s** ppState, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32sc_16sc(IppsFIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32sc_16sc32fc(IppsFIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32fc_16sc(IppsFIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64fc_16sc(IppsFIRState64fc_16sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64fc_32sc(IppsFIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);

```

Case 2: Operation on floating point samples.

```

IppStatus ippsFIRMRInit_32f(IppsFIRState_32f** ppState, const Ipp32f* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor,
    int downPhase, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64f_32f(IppsFIRState64f_32f** ppState, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit_64f(IppsFIRState_64f** ppState, const Ipp64f* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase, const
    Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit_32fc(IppsFIRState_32fc** ppState, const Ipp32fc* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase, const
    Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64fc_32fc(IppsFIRState64fc_32fc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit_64fc(IppsFIRState_64fc** ppState, const Ipp64fc* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor,
    int downPhase, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

```

Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer versions only).
<i>downFactor</i>	Downsampling factor.
<i>downPhase</i>	Phase for downsampled signal.
<i>upFactor</i>	Upsampling factor.
<i>upPhase</i>	Phase for upsampled signal.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $(tapsLen + upFactor - 1) / upFactor$.
<i>ppState</i>	Pointer to the pointer to the FIR state structure.
<i>pBuffer</i>	Pointer to the external buffer for FIR state structure.

Description

The function `ippsFIRMRInit` is declared in the `ipps.h` file. This function initializes a multi-rate FIR filter state structure in the external buffer. The size of this buffer should be computed previously by calling the functions [ippsFIRMTGetStateSize](#). The initialization functions copy the taps from the *tapsLen*-length array *pTaps* into the state structure *ppState*. To scale integer taps, use the *tapsFactor* value.

The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array contents is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

The function initializes the state structure *pState* of a multi-rate filter, that is, a filter that internally upsamples and/or downsamples signals using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the [ippsSampleUp](#) function for more details). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The argument *upPhase* is the parameter, which determines where a non-zero sample lies within the *upFactor*-length block of the upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the [ippsSampleDown](#) function for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

The argument *downPhase* is the parameter, which determines where non-discarded sample lies within a block of upsampled filter response.

If the delay line array *pDlyLine* is not NULL, its length is defined as $(\text{tapsLen} + \text{upFactor} - 1) / \text{upFactor}$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error when <i>upFactor</i> (<i>downFactor</i>) is less than or equal to 0.
<code>ippStsFIRMRPhaseErr</code>	Indicates an error when <i>upPhase</i> (<i>downPhase</i>) is negative, or greater than or equal to <i>upFactor</i> (<i>downFactor</i>).

FIRGetStateSize, FIRMRGetStateSize

Returns the length of the FIR filter state structure.

Syntax

Case 1: Single-rate filter.

```
IppStatus ippsFIRGetStateSize_32s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_32fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_64f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_64fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32s_16s(int tapsLen, int* pBufferSize);
```

```

IppStatus ippsFIRGetStateSize32s_16s32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32sc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32sc_16sc32fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32fc(int tapsLen, int* pBufferSize);

```

Case 2: Multi-rate filter.

```

IppStatus ippsFIRMRGetStateSize_32f(int tapsLen, int upFactor, int downFactor,
    int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_32fc(int tapsLen, int upFactor, int downFactor,
    int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_64f(int tapsLen, int upFactor, int downFactor,
    int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_64fc(int tapsLen, int upFactor, int downFactor,
    int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32s_16s(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32s_16s32f(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32sc_16sc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32sc_16sc32fc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32f_16s(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32fc_16sc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_16s(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);

```



```

IppStatus ippsFIRMRGetStateSize64f_32s(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_32f(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_16sc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_32sc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_32fc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);

```

Parameters

<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>downFactor</i>	Downsampling factor for the multi-rate filter.
<i>upFactor</i>	Upsampling factor for the multi-rate filter.
<i>pBufferSize</i>	Pointer to the computed buffer size value.

Description

The functions `ippsFIRGetStateSize` and `ippsFIRMRGetStateSize` are declared in the `ipps.h` file. These functions compute the size of the external buffer for a single-rate or multi-rate FIR filter state, respectively, and store the result in *pBufferSize*.

To compute the size of a buffer for the single-rate FIR filter state, the number of taps *tapsLen* only need be specified.

To compute the size of a buffer for the multi-rate FIR filter state, the number of taps *tapsLen* and upsampling/downsampling parameters *upFactor* and *downFactor* should be specified.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the [ippsSampleUp](#) function for more details). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the [ippsSampleDown](#) function for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error if <code>nxTapsLen</code> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error if <code>upFactor</code> (<code>downFactor</code>) is less than or equal to 0.

FIRGetTaps

Retrieves the tap values from the FIR filter state structure.

Syntax

```

IppStatus ippFIRGetTaps_32s(const IppsFIRState_32s* pState, Ipp32f* pTaps);
IppStatus ippFIRGetTaps_32f(const IppsFIRState_32f* pState, Ipp32f* pTaps);
IppStatus ippFIRGetTaps_64f(const IppsFIRState_64f* pState, Ipp64f* pTaps);
IppStatus ippFIRGetTaps32f_16s(const IppsFIRState32f_16s* pState, Ipp32f*
    pTaps);
IppStatus ippFIRGetTaps64f_16s(const IppsFIRState64f_16s* pState, Ipp64f*
    pTaps);
IppStatus ippFIRGetTaps64f_32s(const IppsFIRState64f_32s* pState, Ipp64f*
    pTaps);
IppStatus ippFIRGetTaps64f_32f(const IppsFIRState64f_32f* pState, Ipp64f*
    pTaps);
IppStatus ippFIRGetTaps_32fc(const IppsFIRState_32fc* pState, Ipp32fc* pTaps);
IppStatus ippFIRGetTaps_64fc(const IppsFIRState_64fc* pState, Ipp64fc* pTaps);
IppStatus ippFIRGetTaps32fc_16sc(const IppsFIRState32fc_16sc* pState, Ipp32fc*
    pTaps);
IppStatus ippFIRGetTaps64fc_16sc(const IppsFIRState64fc_16sc* pState, Ipp64fc*
    pTaps);
IppStatus ippFIRGetTaps64fc_32sc(const IppsFIRState64fc_32sc* pState, Ipp64fc*
    pTaps);
IppStatus ippFIRGetTaps64fc_32fc(const IppsFIRState64fc_32fc* pState, Ipp64fc*
    pTaps);
IppStatus ippFIRGetTaps32s_16s32f(const IppsFIRState32s_16s* pState, Ipp32f*
    pTaps);

```

```

IppStatus ippsFIRGetTaps32sc_16sc32fc(const IppsFIRState32sc_16sc* pState,
    Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps32s_16s(const IppsFIRState32s_16s* pState, Ipp32s*
    pTaps, int* pTapsFactor);
IppStatus ippsFIRGetTaps32sc_16sc(const IppsFIRState32sc_16sc* pState, Ipp32sc*
    pTaps, int* pTapsFactor);

```

Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>pTaps</i>	Pointer to the array containing the tap values.
<i>pTapsFactor</i>	Pointer to the scale factor for the taps of <code>Ipp32s</code> data type (for integer versions only).

Description

The function `ippsFIRGetTaps` is declared in the `ipps.h` file. This function copies the tap values from the initialized FIR state structure *pState* to the *tapsLen* -length array *pTaps*. To scale integer taps, use the *tapsFactor* value.

Before calling `ippsFIRGetTaps` function, the corresponding filter state structure must be initialized, for example by calling the function [ippsFIRInitAlloc](#) or [ippsFIRMRInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRSetTaps

Sets the tap values in the FIR filter state structure.

Syntax

```

IppStatus ippsFIRSetTaps_32s(const Ipp32s* pTaps, IppsFIRState_32s* pState);
IppStatus ippsFIRSetTaps_32f(const Ipp32f* pTaps, IppsFIRState_32f* pState);
IppStatus ippsFIRSetTaps_64f(const Ipp64f* pTaps, IppsFIRState_64f* pState);

```

```

IppStatus ippsFIRSetTaps32f_16s(const Ipp32f* pTaps, IppsFIRState32f_16s*
    pState);
IppStatus ippsFIRSetTaps64f_16s(const Ipp64f* pTaps, IppsFIRState64f_16s*
    pState);
IppStatus ippsFIRSetTaps64f_32s(const Ipp64f* pTaps, IppsFIRState64f_32s*
    pState);
IppStatus ippsFIRSetTaps64f_32f(const Ipp64f* pTaps, IppsFIRState64f_32f*
    pState);
IppStatus ippsFIRSetTaps_32fc(const Ipp32fc* pTaps, IppsFIRState_32fc* pState);
IppStatus ippsFIRSetTaps_64fc(const Ipp64fc* pTaps, IppsFIRState_64fc* pState);
IppStatus ippsFIRSetTaps32fc_16sc(const Ipp32fc* pTaps, IppsFIRState32fc_16sc*
    pState);
IppStatus ippsFIRSetTaps64fc_16sc(const Ipp64fc* pTaps, IppsFIRState64fc_16sc*
    pState);
IppStatus ippsFIRSetTaps64fc_32sc(const Ipp64fc* pTaps, IppsFIRState64fc_32sc*
    pState);
IppStatus ippsFIRSetTaps64fc_32fc(const Ipp64fc* pTaps, IppsFIRState64fc_32fc*
    pState);
IppStatus ippsFIRSetTaps32s_16s32f(const Ipp32f* pTaps, IppsFIRState32s_16s*
    pState);
IppStatus ippsFIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps, IppsFIRState32sc_16sc*
    pState);
IppStatus ippsFIRSetTaps32s_16s(const Ipp32s* pTaps, IppsFIRState32s_16s* pState,
    int tapsFactor);
IppStatus ippsFIRSetTaps32sc_16sc(const Ipp32sc* pTaps, IppsFIRState32sc_16sc*
    pState, int tapsFactor);

```

Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>pTaps</i>	Pointer to the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of Ipp32s data type (for integer versions only).

Description

The function `ippsFIRSetTaps` is declared in the `ipps.h` file. This function sets new tap values in the previously initialized FIR filter state structure `pState`. New tap values must be specified in the array `pTaps`. The length of the array `pTaps` must be equal to the `tapsLen` parameter value of the initialized filter state.

To scale integer taps, use the `tapsFactor` value.

Before calling `ippsFIRSetTaps` function, the corresponding filter state structure must be initialized, for example by calling the function [ippsFIRInitAlloc](#) or [ippsFIRMRInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRGetDlyLine

Retrieves the delay line contents from the FIR filter state structure.

Syntax

```

IppStatus ippsFIRGetDlyLine_32f(const IppsFIRState_32f* pState, Ipp32f*
    pDlyLine);
IppStatus ippsFIRGetDlyLine_64f(const IppsFIRState_64f* pState, Ipp64f*
    pDlyLine);
IppStatus ippsFIRGetDlyLine32s_16s(const IppsFIRState32s_16s* pState, Ipp16s*
    pDlyLine);
IppStatus ippsFIRGetDlyLine32f_16s(const IppsFIRState32f_16s* pState, Ipp16s*
    pDlyLine);
IppStatus ippsFIRGetDlyLine64f_16s(const IppsFIRState64f_16s* pState, Ipp16s*
    pDlyLine);
IppStatus ippsFIRGetDlyLine64f_32s(const IppsFIRState64f_32s* pState, Ipp32s*
    pDlyLine);

```

```
IppStatus ippsFIRGetDlyLine64f_32f(const IppsFIRState64f_32f* pState, Ipp32f*
    pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine_32fc(const IppsFIRState_32fc* pState, Ipp32fc*
    pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine_64fc(const IppsFIRState_64fc* pState, Ipp64fc*
    pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine32sc_16sc(const IppsFIRState32sc_16sc* pState,
    Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine32fc_16sc(const IppsFIRState32fc_16sc* pState,
    Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine64fc_16sc(const IppsFIRState64fc_16sc* pState,
    Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine64fc_32sc(const IppsFIRState64fc_32sc* pState,
    Ipp32sc* pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine64fc_32fc(const IppsFIRState64fc_32fc* pState,
    Ipp32fc* pDlyLine);
```

Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>pDlyLine</i>	Pointer to the array holding the delay line values.

Description

The function `ippsFIRGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from the state structure *pState* and stores them into *pDlyLine*. The destination array *pDlyLine* contains samples in the reverse order as compared to the order of samples in the source vector.

Before calling `ippsFIRGetDlyLine`, the corresponding filter state structure must be initialized, for example by calling the function [ippsFIRInitAlloc](#) or [ippsFIRMRIInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRSetDlyLine

Sets the delay line contents in the FIR filter state structure.

Syntax

```

IppStatus ippsFIRSetDlyLine_32f(IppsFIRState_32f* pState, const Ipp32f*
    pDlyLine);
IppStatus ippsFIRSetDlyLine_64f(IppsFIRState_64f* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsFIRSetDlyLine32s_16s(IppsFIRState32s_16s* pState, const Ipp16s*
    pDlyLine);
IppStatus ippsFIRSetDlyLine32f_16s(IppsFIRState32f_16s* pState, const Ipp16s*
    pDlyLine);
IppStatus ippsFIRSetDlyLine64f_16s(IppsFIRState64f_16s* pState, const Ipp16s*
    pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32s(IppsFIRState64f_32s* pState, const Ipp32s*
    pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32f(IppsFIRState64f_32f* pState, const Ipp32f*
    pDlyLine);

IppStatus ippsFIRSetDlyLine_32fc(IppsFIRState_32fc* pState, const Ipp32fc*
    pDlyLine);
IppStatus ippsFIRSetDlyLine_64fc(IppsFIRState_64fc* pState, const Ipp64fc*
    pDlyLine);
IppStatus ippsFIRSetDlyLine32sc_16sc(IppsFIRState32sc_16sc* pState, const
    Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine32fc_16sc(IppsFIRState32fc_16sc* pState, const
    Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_16sc(IppsFIRState64fc_16sc* pState, const
    Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32sc(IppsFIRState64fc_32sc* pState, const
    Ipp32sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32fc(IppsFIRState64fc_32fc* pState, const
    Ipp32fc* pDlyLine);

```

Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>pDlyLine</i>	Pointer to the array holding the delay line values.

Description

The function `ippsFIRSetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from *pDlyLine* and stores them into the state structure *pState*. The source array *pDlyLine* must contain samples in the reverse order as compared to the order of samples in the source vector.

Before calling `ippsFIRGetDlyLine` the corresponding filter state structure must be initialized, for example by calling the function [ippsFIRInitAlloc](#) or [ippsFIRMRIInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIROne

Filters a single sample through a FIR filter.

Syntax

Case 1: Integer sample.

```

IppStatus ippsFIROne_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, IppsFIRStates_32s*
    pState, int scaleFactor);
IppStatus ippsFIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsFIRState32s_16s*
    pState, int scaleFactor);
IppStatus ippsFIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsFIRState32f_16s*
    pState, int scaleFactor);
IppStatus ippsFIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsFIRState64f_16s*
    pState, int scaleFactor);
IppStatus ippsFIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, IppsFIRState64f_32s*
    pState, int scaleFactor);

```



```

IppStatus ippsFIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
    IppsFIRState64fc_32sc* pState, int scaleFactor);

```

Case 2: Floating point sample.

```

IppStatus ippsFIROne_32f(Ipp32f src, Ipp32f* pDstVal, IppsFIRState_32f*
    pState);
IppStatus ippsFIROne_64f(Ipp64f src, Ipp64f* pDstVal, IppsFIRState_64f*
    pState);
IppStatus ippsFIROne64f_32f(Ipp32f src, Ipp32f* pDstVal, IppsFIRState64f_32f*
    pState);
IppStatus ippsFIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsFIRState_32fc*
    pState);
IppStatus ippsFIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal, IppsFIRState_64fc*
    pState);
IppStatus ippsFIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsFIRState64fc_32fc*
    pState);

```

Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>src</i>	Input sample to be filtered by the function <code>ippsFIROne</code> .
<i>pDstVal</i>	Pointer to the output sample filtered by the function <code>ippsFIROne</code> .
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsFIROne` is declared in the `ipps.h` file. This function filters a single sample *src* through a single-rate filter and stores the result in *pDstVal*. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted $x(n)$ and the taps are denoted $h(i)$.

The return value $y(n)$ is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Before calling `ippsFIROne`, initialize the filter state by calling [ippsFIRInitAlloc](#). Specify the number of taps `tapsLen`, the tap values in `pTaps`, and the delay line values in `pDlyLine` beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIR

Filters a block of samples through a single-rate or multi-rate FIR filter.

Syntax

Case 1: Not-in-place operation on integer samples.

```
IppStatus ippsFIR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int numIters,
    IppsFIRState_32s* pState, int scaleFactor);
IppStatus ippsFIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
    IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
    IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
    IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int numIters,
    IppsFIRState64f_32s* pState, int scaleFactor);
IppStatus ippsFIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, IppsFIRState32fc_16sc* pState, int scaleFactor);
```

```

IppStatus ippsFIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int
    numIters, IppsFIRState64fc_32sc* pState, int scaleFactor);

```

Case 2 Not-in-place operation on floating point samples.

```

IppStatus ippsFIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    IppsFIRState_32f* pState);
IppStatus ippsFIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
    IppsFIRState_64f* pState);
IppStatus ippsFIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
    IppsFIRState_32fc* pState);
IppStatus ippsFIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int numIters,
    IppsFIRState_64fc* pState);
IppStatus ippsFIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    IppsFIRState64f_32f* pState);
IppStatus ippsFIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
    IppsFIRState64fc_32fc* pState);

```

Case 3 In-place operation on integer samples.

```

IppStatus ippsFIR_32s_ISfs(Ipp32s* pSrcDst, int numIters, IppsFIRState_32s*
    pState, int scaleFactor);
IppStatus ippsFIR32s_16s_ISfs(Ipp16s* pSrcDst, int numIters,
    IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIR32f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
    IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
    IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_32s_ISfs(Ipp32s* pSrcDst, int numIters,
    IppsFIRState64f_32s* pState, int scaleFactor);
IppStatus ippsFIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
    IppsFIRState64fc_32sc* pState, int scaleFactor);

```

Case 4 In-place operation on floating point samples.

```

IppStatus ippsFIR_32f_I(Ipp32f* pSrcDst, int numIters, IppsFIRState_32f*
    pState);
IppStatus ippsFIR_64f_I(Ipp64f* pSrcDst, int numIters, IppsFIRState_64f*
    pState);
IppStatus ippsFIR64f_32f_I(Ipp32f* pSrcDst, int numIters, IppsFIRState64f_32f*
    pState);
IppStatus ippsFIR_32fc_I(Ipp32fc* pSrcDst, int numIters, IppsFIRState_32fc*
    pState);
IppStatus ippsFIR_64fc_I(Ipp64fc* pSrcDst, int numIters, IppsFIRState_64fc*
    pState);
IppStatus ippsFIR64fc_32fc_I(Ipp32fc* pSrcDst, int numIters, IppsFIRState64fc_32fc*
    pState);

```

Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>pSrc</i>	Pointer to the input array to be filtered by the function <code>ippsFIR</code> .
<i>pDst</i>	Pointer to the output array filtered by the function <code>ippsFIR</code> .
<i>pSrcDst</i>	Pointer to the input and output array (for the in-place operation) to be filtered by the function <code>ippsFIR</code> .
<i>numIters</i>	Parameter associated with the number of samples to be filtered by the function <code>ippsFIR</code> . For single-rate filters, the <i>numIters</i> samples in the input array are filtered and the resulting <i>numIters</i> samples are stored in the output array. For multi-rate filters, the (<i>numIters</i> * <i>downFactor</i>) samples in the input array are filtered and the resulting (<i>numIters</i> * <i>upFactor</i>) samples are stored in the output array.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsFIR` is declared in the `ipps.h` file. This function filters an input array *pSrc* or *pSrcDst* through a single-rate or multi-rate filter, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*.

For single-rate filters, the *numIters* samples in the array *pSrc* or *pSrcDst* are filtered, and the resulting *numIters* samples are stored in the array *pDst* or *pSrcDst*. The results are identical to *numIters* consecutive calls to `ippsFIROne`.

In the following definition of the FIR filter, the sample to be filtered is denoted $x(n)$, the taps are denoted $h(i)$, and the return value is $y(n)$.

The return value $y(n)$ is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \leq n < numIters$$

After the function has performed calculations, it updates the delay line values stored in the state. For a single-rate filter, the *numIters* parameter defines the length of the source and destination arrays.

For multi-rate filters, *ippsFIR* filters (*numIters*downFactor*) input samples and stores the resulting (*numIters*upFactor*) samples in the output array. The multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps. Thus, the function does not create an internal (*upFactor*srcLen*)-size buffer to store the upsampling result.

Before calling *ippsFIR*, initialize the filter state by calling either [ippsFIRInitAlloc](#) for single-rate filter, or [ippsFIRMRInitAlloc](#) for multi-rate filter. Specify the number of taps *tapsLen*, the tap values in *pTaps*, and the delay line values in *pDlyLine* beforehand.

[Example 6-3](#) illustrates single-rate filtering with the function *ippsFIR_32f*.

Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when one of the specified pointers is NULL.
<i>ippStsSizeErr</i>	Indicates an error when <i>numIters</i> is less or equal to 0.
<i>ippStsContextMatchErr</i>	Indicates an error when the state identifier is incorrect.

Example 6-3 Single-Rate Filtering with the ippsFIR Function

```
IppStatus fir(void) {
    #undef NUMITERS
    #define NUMITERS 150
        int n;
        IppStatus status;
        IppsIIRState_32f *ictx;
        IppsFIRState_32f *fctx;
        Ipp32f *x = ippsMalloc_32f(NUMITERS),
            *y = ippsMalloc_32f(NUMITERS),
            *z = ippsMalloc_32f(NUMITERS);
        const float taps[] = {
            0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
            0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
        };
        for (n = 0; n < NUMITERS; ++n) x[n] = (float) sin(IPP_2PI * n * 0.2);
        ippsIIRInitAlloc_32f( &ictx, taps, 10, NULL );
        ippsFIRInitAlloc_32f( &fctx, taps, 11, NULL );
        status = ippsIIR_32f( x, y, NUMITERS, ictx );
        printf_32f("IIR 32f output + 120 =", y+120, 5, status);
        ippsIIRFree_32f(ictx);
        status = ippsFIR_32f( x, z, NUMITERS, fctx );
        printf_32f("FIR 32f output + 120 =", z+120, 5, status);
        ippsFIRFree_32f(fctx);
        ippsFree(z);
        ippsFree(y);
        ippsFree(x);
        return status;
    }
```

Output:

```
IIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
```

```
FIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
```

Matlab* Analog:

```
>> F = 0.2; N = 150; n = 0:N-1; x = sin(2*pi*n*F);
y = filter(fir1(10,0.15),1,x); y(121:125)
```

FIROne_Direct

Directly filters a single sample through a FIR filter.

Syntax

Case 1: Not-in-place operation on integer sample.

```

IppStatus ippsFIROne_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const Ipp16s*
    pTapsQ15, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIROne32f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const
    Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIROne64f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const
    Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIROne64f_Direct_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, const
    Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIROne32fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal, const
    Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIROne64fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal, const
    Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIROne64fc_Direct_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal, const
    Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIROne32s_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const
    Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32sc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

```

Case 2: Not-in-place operation on floating point sample.

```

IppStatus ippsFIROne_Direct_32f(Ipp32f src, Ipp32f* pDstVal, const Ipp32f*
    pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);

```

```

IppStatus ippsFIROne_Direct_64f(Ipp64f src, Ipp64f* pDstVal, const Ipp64f*
    pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal, const Ipp32fc*
    pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64fc(Ipp64fc src, Ipp64fc* pDstVal, const Ipp64fc*
    pTaps, int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne64f_Direct_32f(Ipp32f src, Ipp32f* pDstVal, const Ipp64f*
    pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne64fc_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal, const
    Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);

```

Case 3: In-place operation on integer sample.

```

IppStatus ippsFIROne_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp16s* pTaps,
    int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32f_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp32f*
    pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp64f*
    pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_32s_ISfs(Ipp32s* pSrcDstVal, const Ipp64f*
    pTaps, int tapsLen, Ipp32s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal, const Ipp32fc*
    pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal, const Ipp64fc*
    pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDstVal, const Ipp64fc*
    pTaps, int tapsLen, Ipp32sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32s_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp32s*
    pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine, int* pDlyLineIndex,
    int scaleFactor);
IppStatus ippsFIROne32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal, const Ipp32sc*
    pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine, int* pDlyLineIndex,
    int scaleFactor);

```

Case 4: In-place operation on floating point sample.

```

IppStatus ippsFIROne_Direct_32f_I(Ipp32f* pSrcDstVal, const Ipp32f* pTaps, int
    tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64f_I(Ipp64f* pSrcDstVal, const Ipp64f* pTaps, int
    tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);

```



```

IppStatus ippsFIROne_Direct_32fc_I(Ipp32fc* pSrcDstVal, const Ipp32fc* pTaps,
    int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64fc_I(Ipp64fc* pSrcDstVal, const Ipp64fc* pTaps,
    int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne64f_Direct_32f_I(Ipp32f* pSrcDstVal, const Ipp64f* pTaps,
    int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne64fc_Direct_32fc_I(Ipp32fc* pSrcDstVal, const Ipp64fc*
    pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);

```

Parameters

<i>src</i>	Input sample to be filtered by the function.
<i>pDstVal</i>	Pointer to the output sample filtered by the function.
<i>pSrcDstVal</i>	Pointer to the input and output sample for in-place operation.
<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>pTapsQ15</i>	Pointer to the array containing the tap values, represented in Q0.15 format. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of Ipp32s data type.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * tapsLen$. Note that the delay line length is different than that for FIR filters using state structure.
<i>pDlyLineIndex</i>	Pointer to the current delay line index.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsFIROne_Direct` is declared in the `ipps.h` file. This function directly filters a single sample *src* or *pSrcDstVal* through a single-rate filter and stores the result in *pDstVal* or *pSrcDstVal*.

The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps, the *tapsFactor* value is used.

The set of *tapsLen* input samples is copied twice to the $2 * tapsLen$ -length array *pDlyLine*. Double length of the delay line in direct FIR filters is used to improve filter performance by decreasing the number of sample copyings. The current delay line index is specified in the *pDlyLineIndex*.

The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted $x(n)$ and the taps are denoted $h(i)$.

The return value $y(n)$ is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.

FIR_Direct

Directly filters a block of samples through a single-rate FIR filter.

Syntax

Case 1: Not-in-place operation on integer samples.

```

IppStatus ippsFIR_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp16s* pTapsQ15, int tapsLen, Ipp16s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64f_Direct_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

```

```

IppStatus ippsFIR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32s_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s*
    pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc*
    pDlyLine, int* pDlyLineIndex, int scaleFactor);

```

Case 2: Not-in-place operation on floating point samples.

```

IppStatus ippsFIR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
    const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
    const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int numIters,
    const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int*
    pDlyLineIndex);

```

Case 3: In-place operation on integer samples.

```

IppStatus ippsFIR_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const Ipp16s*
    pTapsQ15, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

```

```

IppStatus ippsFIR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);

IppStatus ippsFIR32s_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

```

Case 4: In-place operation on floating point samples.

```

IppStatus ippsFIR_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const Ipp32f*
    pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_64f_I(Ipp64f* pSrcDst, int numIters, const Ipp64f*
    pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const Ipp32fc*
    pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters, const Ipp64fc*
    pTaps, int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const Ipp64f*
    pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);

```

Parameters

pSrc Pointer to the input array to be filtered.

<i>pDst</i>	Pointer to the output array.
<i>pSrcDst</i>	Pointer to the input and output array for the in-place operation.
<i>numIters</i>	Number of samples in the input array to be filtered.
<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>pTapsQ15</i>	Pointer to the array containing the tap values, represented in Q0.15 format. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * \textit{tapsLen}$. Note that the delay line length is different than that for FIR filters using state structure.
<i>pDlyLineIndex</i>	Pointer to the current delay line index.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsFIR_Direct` is declared in the `ipps.h` file. This function filters an input array *pSrc* or *pSrcDst* containing *numIters* samples through a single-rate filter, and stores the resulting *numIters* samples in *pDst* or *pSrcDst*, respectively. The results are identical to *numIters* consecutive calls to `ippsFIROne_Direct`.

The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used.

The set of *tapsLen* input samples is copied to the $2 * \textit{tapsLen}$ -length array *pDlyLine*. Double length of the delay line in direct FIR filters is used to improve filter performance by decreasing the number of sample copyings. The current delay line index is specified in the *pDlyLineIndex*.

The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

In the following definition of the FIR filter, the sample to be filtered is denoted $x(n)$, the taps are denoted $h(i)$, and the return value is $y(n)$.

The return value $y(n)$ is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{\textit{tapsLen} - 1} h(i) \cdot x(n - i), \quad 0 \leq n < \textit{numIters}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>numIters</i> is less than or equal to 0.

FIRMR_Direct

Directly filters a block of samples through a multi-rate FIR filter.

Syntax

Case 1: Not-in-place operation on integer samples.

```

IppStatus ippsFIRMR32f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64f_Direct_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32sc* pDlyLine, int scaleFactor);

```

```

IppStatus ippsFIRMR32s_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp16s* pDlyLine, int
    scaleFactor);

IppStatus ippsFIRMR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp16sc* pDlyLine, int
    scaleFactor)

```

Case 2: Not-in-place operation on floating point samples.

```

IppStatus ippsFIRMR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
    int downPhase, Ipp32f* pDlyLine);

IppStatus ippsFIRMR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32f* pDlyLine);

IppStatus ippsFIRMR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
    const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
    int downPhase, Ipp64f* pDlyLine);

IppStatus ippsFIRMR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
    numIters, const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32fc* pDlyLine);

IppStatus ippsFIRMR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32fc* pDlyLine);

IppStatus ippsFIRMR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp64fc* pDlyLine);

```

Case 3: In-place operation on integer samples.

```

IppStatus ippsFIRMR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32s* pDlyLine, int scaleFactor);

```

```

IppStatus ippsFIRMR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp16sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp16sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR32s_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);

```

Case 4: In-place operation on floating point samples.

```

IppStatus ippsFIRMR_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const Ipp32f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32f* pDlyLine);

IppStatus ippsFIRMR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32f* pDlyLine);

IppStatus ippsFIRMR_Direct_64f_I(Ipp64f* pSrcDst, int numIters, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp64f* pDlyLine);

IppStatus ippsFIRMR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32fc* pDlyLine);

IppStatus ippsFIRMR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32fc* pDlyLine);

IppStatus ippsFIRMR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp64fc* pDlyLine);

```

Parameters

pSrc

Pointer to the input array to be filtered.

<i>pDst</i>	Pointer to the output array.
<i>pSrcDst</i>	Pointer to the input and output array for the in-place operation.
<i>numIters</i>	Parameter associated with the number of samples to be filtered by the function. The $(numIters * downFactor)$ samples of the input array are filtered and the resulting $(numIters * upFactor)$ samples are stored in the output array
<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <i>Ipp32s</i> data type.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $(tapsLen + upFactor - 1) / upFactor$.
<i>upFactor</i>	Factor for upsampling the multi-rate signals.
<i>downFactor</i>	Factor for downsampling the multi-rate signals.
<i>upPhase</i>	Phase for upsampling the multi-rate signals.
<i>downPhase</i>	Phase for downsampling the multi-rate signals.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsFIRMR_Direct` is declared in the `ipps.h` file. This function filters an input array *pSrc* or *pSrcDst* through a multi-rate filter, and stores the resulting samples in *pDst* or *pSrcDst*, respectively. The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used. The array *pDlyLine* specifies the delay line values. The input array contains $(numIters * downFactor)$ samples, and the output array stores the resulting $(numIters * upFactor)$ samples.

The multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the [ippsSampleUp](#) function for more details). That is, *upFactor*-1 zeros are inserted between each sample of input signal.

The argument *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see description of the [ippsSampleDown](#) function for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The argument *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response. The length of the delay line array *pDlyLine* is defined as $(\text{tapsLen} + \text{upFactor} - 1) / \text{upFactor}$.

The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>numIters</i> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error when <i>upFactor</i> (<i>downFactor</i>) is less than or equal to 0.
<code>ippStsFIRMRPhaseErr</code>	Indicates an error when <i>upPhase</i> (<i>downPhase</i>) is negative, or greater than or equal to <i>upFactor</i> (<i>downFactor</i>).

FIRSparseInit

Initializes a sparse FIR filter structure.

Syntax

```

IppStatus ippsFIRSparseInit_32f(IppsFIRSparseState_32f** ppState,
    const Ipp32f* pNZTaps, const Ipp32s* pNZTapPos, int nzTapsLen, const
    Ipp32f* pDlyLine, Ipp8u* pBuffer);

```

Parameters

<i>pNZTaps</i>	Pointer to the array containing the non-zero tap values. The number of elements in the array is <i>nzTapsLen</i> .
----------------	--

<i>pNZTapPos</i>	Pointer to the array containing positions of the non-zero tap values. The number of elements in the array is <i>nzTapsLen</i> .
<i>nzTapsLen</i>	Number of elements in the array with non-zero tap values.
<i>pDlyLine</i>	Pointer to the array containing the delay line values.
<i>ppState</i>	Double pointer to the sparse FIR state structure.
<i>pBuffer</i>	Pointer to the external buffer for the sparse FIR state structure.

Description

The functions `ippsFIRSparseInit` is declared in the `ipps.h` file. This function initializes a sparse FIR filter state structure *pState* in the external buffer *pBuffer*. The size of this buffer should be computed previously by calling the function [ippsFIRSparseGetStateSize](#). The initialization function copies the values of filter coefficients from the array *pNZTaps* containing *nzTapsLen* non-zero taps and their positions from the array *pNZTapPos* into the state structure *pState*.

The array *pDlyLine* specifies the delay line values. The number of elements in this array is *pNZTapPos[nzTapsLen - 1]*. If the pointer to the array *pDlyLine* is not NULL, the array contents are copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.



CAUTION. The values of *nzTapsLen* and *pNZTapPos[nzTapsLen - 1]* should be equal to those specified in the [ippsFIRSparseGetStateSize](#) function.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error if one of the pointers <i>ppState</i> , <i>pNZTaps</i> , <i>pNZTapPos</i> , or <i>pBuffer</i> is NULL.
<code>ippsStsFIRLenErr</code>	Indicates an error if <i>nxTapsLen</i> is less than or equal to 0.
<code>ippsStsSparseErr</code>	Indicates an error if positions of the non-zero taps are not in ascending order, or are negative or repetitive.

FIRSparseGetStateSize

Computes the size of the external buffer for the sparse FIR filter structure.

Syntax

```
ippStatus ippFIRSparseGetStateSize_32f(int nzTapsLen, int order, int*
    pStateSize);
```

Parameters

<i>nzTapsLen</i>	Number of elements in the array containing the non-zero tap values.
<i>order</i>	Order of the sparse FIR filter.
<i>pStateSize</i>	Pointer to the computed value of the external buffer.

Description

The function `ippFIRSparseGetStateSize` is declared in the `ipps.h` file. This function computes the size of the external buffer for a sparse FIR filter structure that is required for the function [ippsFIRSparseInit](#). Computation is based on the specified number of non-zero filter coefficients *nzTapsLen* and filter order *order* that is equal to the number of elements in the delay line `pNZTapPos[nzTapsLen - 1]` (see description of the function [ippsFIRSparseInit](#)). The result value is stored in the *pStateSize*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pStateSize</i> pointer is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error if <i>nzTapsLen</i> or <i>order</i> is less than or equal to 0; or <i>nzTapsLen</i> is more than <i>order</i> .

FIRSparse

Filters a block of samples through a sparse FIR filter.

Syntax

```
IppStatus ippsFIRSparse_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    IppsFIRSparseState_32f* pState);
```

Parameters

<i>pState</i>	Pointer to the sparse FIR filter state structure.
<i>pSrc</i>	Pointer to the input array.
<i>pDst</i>	Pointer to the output array.
<i>len</i>	Number of elements that will be filtered.

Description

The function `ippsFIRSparse` is declared in the `ipps.h` file. This function applies the sparse FIR filter to the *len* elements of the input array *pSrc*, and stores the results in *pDst*. The filter parameters - the number of non-zero taps *nzTapsLen*, their values *pNZTaps* and their positions *pNZTapPos*, and the delay line values *pDlyLine* - are specified in the sparse FIR filter structure *pState* that should be previously initialized by calling the function [ippsFIRSparseInit](#).

In the following definition of the sparse FIR filter, the sample to be filtered is denoted $x(n)$, the non-zero taps are denoted $pNZTaps(i)$, their positions are denoted $pNZTapPos(i)$ and the return value is $y(n)$.

The return value $y(n)$ is defined by the formula for a sparse FIR filter:

$$y(n) = \sum_{i=0}^{nzTapsLen-1} pNZTaps(i) \cdot x(n - pNZTapPos(i)), \quad 0 \leq n < len$$

After the function has performed calculations, it updates the delay line values stored in the state.

[Example 6-4](#) shows how to use the sparse FIR filter functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less or equal to 0.

Example 6-4 Using the Sparse FIR Filter Functions

```
int buflen;
Ipp8u *buf;
int nzTapsLen = 5;//number of non-zero taps
Ipp32f nzTaps [] = {0.5, 0.4, 0.3, 0.2, 0.1};//non-zero taps values
Ipp32s nzTapsPos[] = {0, 10, 20, 30, 40};//non-zero tap positions
IppsFIRSparseState_32f* firState;
Ipp32f *src, *dst;

/* ..... */

ippsFIRSparseGetStateSize_32f(nzTapsLen, nzTapsPos [nzTapsLen - 1], &buflen);
buf = ippsMalloc_8u(buflen);
ippsFIRSparseInit_32f(&firState, nzTaps, nzTapsPos, nzTapsLen, NULL, buf);

/* .... initializing src somehow .... */

ippsFIRSparse_32f(src, dst, len, firState);

/*dst[i]=src[i]*0.5+src[i-10]*0.4+src[i-20]*0.3+src[i-30]*0.2+src[i-40]*0.1*/

/* ..... */

ippsFree(buf);
```

FIR Filter Coefficient Generating Functions

The functions described in this section compute coefficients (tap values) for different FIR filters by windowing the ideal infinite filter coefficients.

FIRGenLowpass

Computes lowpass FIR filter coefficients.

Syntax

```
IppStatus ippSFIRGenLowpass_64f(Ipp64f rFreq, Ipp64f* taps, int tapsLen,  
    IppWinType winType, IppBool doNormal);
```

Parameters

<i>rFreq</i>	Normalized cut-off frequency, should be in the range (0, 0.5).
<i>pTaps</i>	Pointer to the array where computed tap values are stored. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values; should be equal or greater than 5.
<i>winType</i>	Values of this argument specify what type of window will be used in computations. The <i>winType</i> must have one of the following values: <i>ippWinBartlett</i> Bartlett window; <i>ippWinBlackman</i> Blackman window; <i>ippWinHamming</i> Hamming window; <i>ippWinHann</i> Hann window.
<i>doNormal</i>	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The <i>doNormal</i> must have one of the following values: <i>ippTrue</i> The function computes normalized sequence of coefficients. <i>ippFalse</i> The function computes non-normalized sequence of coefficients.

Description

The function `ippsFIRGenLowpass` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for lowpass FIR filter with the cut-off frequency *rFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see [Windowing Functions](#). The computed coefficients are stored in the array *pTaps*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTaps</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when the <i>tapsLen</i> is less than 5, or <i>rFreq</i> is out of range.

FIRGenHighpass

Computes highpass FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenHighpass_64f(Ipp64f rFreq, Ipp64f* taps, int tapsLen,
    IppWinType winType, IppBool doNormal);
```

Parameters

<i>rFreq</i>	Normalized cut-off frequency, should be in the range (0, 0.5).
<i>pTaps</i>	Pointer to the array where computed tap values are stored. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values; should be equal or greater than 5.
<i>winType</i>	Values of this argument specify what type of window will be used in computations. The <i>winType</i> must have one of the following values: <code>ippWinBartlett</code> Bartlett window; <code>ippWinBlackman</code> Blackman window;

	<code>ippWinHamming</code>	Hamming window;
	<code>ippWinHann</code>	Hann window.
<code>doNormal</code>	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The <code>doNormal</code> must have one of the following values:	
	<code>ippTrue</code>	The function computes normalized sequence of coefficients.
	<code>ippFalse</code>	The function computes non-normalized sequence of coefficients.

Description

The function `ippsFIRGenHighpass` is declared in the `ipps.h` file. This function computes `tapsLen` coefficients for highpass FIR filter the cut-off frequency `rFreq` by windowing the ideal infinite filter coefficients. The parameter `winType` specifies the type of the window. For more information on window types used by the function, see [Windowing Functions](#). The computed coefficients are stored in the array `pTaps`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pTaps</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when the <code>tapsLen</code> is less than 5, or <code>rFreq</code> is out of the range.

FIRGenBandpass

Computes bandpass FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenBandpass_64f(Ipp64f rLowFreq, Ipp64f rHighFreq, Ipp64f*
    pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

Parameters

<code>rLowFreq</code>	Normalized low cut-off frequency, should be in the range (0, 0.5) and less than <code>rHighFreq</code> .
-----------------------	--

<i>rHighFreq</i>	Normalized high cut-off frequency, should be in the range (0, 0.5) and greater than <i>rLowFreq</i> .
<i>pTaps</i>	Pointer to the array where computed tap values are stored. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values; should be equal or greater than 5.
<i>winType</i>	Values of this argument specify what type of window will be used in computations. The <i>winType</i> must have one of the following values: <code>ippWinBartlett</code> Bartlett window; <code>ippWinBlackman</code> Blackman window; <code>ippWinHamming</code> Hamming window; <code>ippWinHann</code> Hann window.
<i>doNormal</i>	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The <i>doNormal</i> must have one of the following values: <code>ippTrue</code> The function computes normalized sequence of coefficients. <code>ippFalse</code> The function computes non-normalized sequence of coefficients.

Description

The function `ippsFIRGenBandpass` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for bandpass FIR filter with the cut-off frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see [Windowing Functions](#). The computed coefficients are stored in the array *pTaps*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTaps</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when the <i>tapsLen</i> is less than 5, or <i>rLowFreq</i> is greater than or equal to <i>rHighFreq</i> , or one of the frequency parameters <i>rLowFreq</i> and <i>rHighFreq</i> is out of the range.

FIRGenBandstop

Computes bandstop FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenBandstop_64f(Ipp64f rLowFreq, Ipp64f rHighFreq, Ipp64f*
    pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

Parameters

<i>rLowFreq</i>	Normalized low cut-off frequency, should be in the range (0, 0.5) and less than <i>rHighFreq</i> .
<i>rHighFreq</i>	Normalized high cut-off frequency, should be in the range (0, 0.5) and greater than <i>rLowFreq</i> .
<i>pTaps</i>	Pointer to the array where computed tap values are stored. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values, should be equal or greater than 5.
<i>winType</i>	Values of this argument specify what type of window will be used in computations. The <i>winType</i> must have one of the following values: <i>ippWinBartlett</i> Bartlett window; <i>ippWinBlackman</i> Blackman window; <i>ippWinHamming</i> Hamming window; <i>ippWinHann</i> Hann window.
<i>doNormal</i>	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The <i>doNormal</i> must have one of the following values: <i>ippTrue</i> The function computes normalized sequence of coefficients. <i>ippFalse</i> The function computes non-normalized sequence of coefficients.

Description

The function `ippsFIRGenBandstop` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for bandstop FIR filter with the cut-off frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see [Windowing Functions](#). The computed coefficients are stored in the array *pTaps*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTaps</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when the <i>tapsLen</i> is less than 5, or <i>rLowFreq</i> is greater than or equal to <i>rHighFreq</i> , or one of the frequency parameters <i>rLowFreq</i> and <i>rHighFreq</i> is out of the range.

Single-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a single-rate FIR LMS filter
- get and set the delay line values
- get the filter coefficients (taps) values
- perform filtering
- free dynamic memory allocated for the functions state

To use the single-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call [ippsFIRLMSInitAlloc](#) to allocate memory and initialize a single-rate FIR LMS filter.
2. Call [ippsFIRLMSOne_Direct](#) to make one iteration of FIR filter taps fitting with one input sample and/or call `ippsFIRLMS` to fit taps with a block of consecutive input samples.
3. Call [ippsFIRLMSGetTaps](#) to get the filter coefficients (taps). Call [ippsFIRLMSGetDlyLine](#) and [ippsFIRLMSSetDlyLine](#) to get and set the values in the delay line.
4. Call [ippsFIRLMSFree](#) to release dynamic memory associated with the FIR LMS filter.

FIRLMSInitAlloc

Allocates memory and initializes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.

Syntax

```
IppStatus ippsFIRLMSInitAlloc_32f(IppsFIRLMSState_32f** ppState, const Ipp32f* pTaps, int tapsLen, const Ipp32f* pDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSInitAlloc32f_16s(IppsFIRLMSState32f_16s** ppState, const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine, int dlyLineIndex);
```

Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>pDlyLine</i>	Pointer to the array holding the delay line values. The number of elements in the array is $2 * \textit{tapsLen}$.
<i>dlyLineIndex</i>	Current index of the delay line.
<i>ppState</i>	Pointer to the pointer to the state structure.

Description

The function `ippsFIRLMSInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a single-rate FIR LMS filter state. The function `ippsFIRLMSInitAlloc` copies the taps from the *tapsLen*-length array *pTaps* into the state structure *ppState*. The $2 * \textit{tapsLen}$ -length array *pDlyLine* specifies the delay line values. The current index of the delay line *pDlyLine* is defined by *dlyLineIndex*. If the pointer to the array *pDlyLine* or *pTaps* is NULL, then the corresponding values of the state structure are initialized to 0.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSFree

Closes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.

Syntax

```
IppStatus ippFIRLMSFree_32f(IppsFIRLMSState_32f* pState);
IppStatus ippFIRLMSFree32f_16s(IppsFIRLMSState32f_16s* pState);
```

Parameters

pState Pointer to the FIR LMS filter state structure to be closed.

Description

The function `ippFIRLMSFree` is declared in the `ipps.h` file. This function closes the FIR LMS filter state by freeing all memory associated with a filter state created by [ippFIRLMSInitAlloc](#). Call `ippFIRLMSFree` after filtering is completed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSGetTaps

Retrieves the tap values from the FIR LMS filter.

Syntax

```
IppStatus ippFIRLMSGetTaps_32f(const IppsFIRLMSState_32f* pState, Ipp32f*
    pOutTaps);
IppStatus ippFIRLMSGetTaps32f_16s(const IppsFIRLMSState32f_16s* pState,
    Ipp32f* pOutTaps);
```

Parameters

<i>pState</i>	Pointer to the FIR LMS filter state structure.
<i>pOutTaps</i>	Pointer to the array holding copies of the taps.

Description

The function `ippsFIRLMSGetTaps` is declared in the `ipps.h` file. This function copies the taps from the state structure *pState* to the *tapsLen*-length array *pOutTaps*. To set new taps in the state structure, create a new state using the function `ippsFIRLMSInitAlloc`.

Before calling the function `ippsFIRLMSGetTaps` the filter state structure must be initialized by the function [`ippsFIRLMSInitAlloc`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSGetDlyLine

Retrieves the delay line contents from the FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSGetDlyLine_32f(const IppsFIRLMSState_32f* pState, Ipp32f*
    pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSGetDlyLine32f_16s(const IppsFIRLMSState32f_16s* pState,
    Ipp16s* pDlyLine, int* pDlyLineIndex);
```

Parameters

<i>pState</i>	Pointer to the FIR LMS filter state structure.
<i>pDlyLine</i>	Pointer to the <i>tapsLen</i> -length array holding the delay line values.
<i>pDlyLineIndex</i>	Pointer to the array to store the current delay line index copied from the filter state structure.

Description

The function `ippsFIRLMSSetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values and the current delay line index from the state structure `pState`, and stores them into `pDlyLine` and `pDlyLineIndex`, respectively.

Before calling the function `ippsFIRLMSSetDlyLine` the filter state structure must be initialized by the function [`ippsFIRLMSInitAlloc`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSSetDlyLine

Sets the delay line contents in the FIR LMS filter.

Syntax

```

IppStatus ippsFIRLMSSetDlyLine_32f(const IppsFIRLMSState_32f* pState, Ipp32f*
    pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine32f_16s(const IppsFIRLMSState32f_16s* pState,
    Ipp16s* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine_32f(IppsFIRLMSState_32f* pState, const Ipp32f*
    pDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine32f_16s(IppsFIRLMSState32f_16s* pState, const
    Ipp16s* pDlyLine, int dlyLineIndex);

```

Parameters

<code>pState</code>	Pointer to the FIR LMS filter state structure.
<code>pDlyLine</code>	Pointer to the <code>tapLen</code> -length array holding the delay line values.
<code>dlyLineIndex</code>	Initial index of the delay line to be stored in the filter state structure <code>pState</code> .

Description

The function `ippsFIRLMSSetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from `pDlyLine`, and the current delay line index from `dlyLineIndex`, and stores them into the state structure `pState`.

Before calling the function `ippsFIRLMSSetDlyLine` the filter state structure must be initialized by the function [`ippsFIRLMSInitAlloc`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMS

Filters an array through a FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMS_32f(const Ipp32f* pSrc, const Ipp32f* pRef, Ipp32f* pDst,
    int len, float mu, IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMS32f_16s(const Ipp16s* pSrc, const Ipp16s* pRef, Ipp16s*
    pDst, int len, float mu, IppsFIRLMSState32f_16s* pState);
```

Parameters

<code>pState</code>	Pointer to the FIR LMS filter state structure.
<code>pSrc</code>	Pointer to the input sample to be filtered.
<code>pRef</code>	Pointer to the reference signal
<code>pDst</code>	Pointer to the output signal
<code>len</code>	Number of elements in the array.
<code>mu</code>	Adaptation step.

Description

The function `ippsFIRLMS` is declared in the `ipps.h` file. This function filters an input array `pSrc` using an adaptive FIR LMS filter.

Each of `len` iterations performed by the function consists of two main procedures. First, `ippsLMS` filters the current sample of the input signal `pSrc` and stores the result in `pDst`. Next, the function updates the current taps using the reference signal `pRef`, the computed result signal `pDst`, and the adaptation step `mu`.

The filtering procedure can be described as a FIR filter operation:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Here the input sample to be filtered is denoted by $x(n)$, the taps are denoted by $h(i)$, and $y(n)$ is the return value.

The function updates the filter coefficients that are stored in the filter state structure `pState`. Updated filter coefficients are defined as

$$h_{n+1}(i) = h_n(i) + 2 \cdot \mu \cdot errVal \cdot x(n-i)$$

where $h_{n+1}(i)$ denotes new taps, $h_n(i)$ denotes initial taps, μ and $errVal$ are the adaptation step and adaptation error value, respectively. An adaptation error value $errVal$ is computed inside the function as the difference between the output and reference signals.

Before using `ippsFIRLMS`, initialize the `pState` structure by calling the [`ippsFIRLMSInitAlloc`](#) function.

[Example 6-5](#) illustrates the use of the function `ippsFIRLMS_32f` to filter a signal sample.

Example 6-5 Filtering with the `ippsFIRLMS` Function

```
IppStatus firlms(void) {
    IppStatus st;
    Ipp32f taps = 0, x[LEN], y[LEN], mu = 0.03f;
    IppsFIRLMSState_32f* ctx;
    int i;
    /// no taps and no delay line from outside
    ippsFIRLMSInitAlloc_32f( &ctx, 0, 1, 0, 0 );
    /// make a const signal of amplitude 1 and noise it
    for(i=0; i<LEN; ++i) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
    st = ippsFIRLMS_32f( x, x+1, y, LEN-1, mu, ctx);
    /// get FIR LMS tap, it must be near to 1
    ippsFIRLMSGetTaps_32f(ctx, &taps);
    ippsFIRLMSFree_32f(ctx);
    printf_32f("FIR LMS tap fitted =", &taps, 1, st);
    return st;
}
```

Output:

FIR LMS tap adapted = 0.986842

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSOne_Direct

Filters a single sample through a FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSOne_Direct_32f(Ipp32f src, Ipp32f refVal, Ipp32f* pDstVal,
    Ipp32f* pTapsInv, int tapsLen, float mu, Ipp32f* pDlyLine, int*
    pDlyLineIndex);

IppStatus ippsFIRLMSOne_Direct32f_16s(Ipp16s src, Ipp16s refVal, Ipp16s*
    pDstVal, Ipp32f* pTapsInv, int tapsLen, float mu, Ipp16s* pDlyLine, int*
    pDlyLineIndex);

IppStatus ippsFIRLMSOne_DirectQ15_16s(Ipp16s src, Ipp16s refVal, Ipp16s*
    pDstVal, Ipp32s* pTapsInv, int tapsLen, int muQ15, Ipp16s* pDlyLine, int*
    pDlyLineIndex);
```

Parameters

<i>src</i>	Input sample to be filtered.
<i>pDstVal</i>	Pointer to the output sample.
<i>refVal</i>	Reference signal sample.
<i>pTapsInv</i>	Pointer to the array containing the FIR filter taps to be adapted. The tap values are stored in the array in the inverse order.
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>pDlyLine</i>	Pointer to the array holding the delay line values.
<i>pDlyLineIndex</i>	Pointer to the current index of the delay line.
<i>mu</i>	Adaptation step.
<i>muQ15</i>	Integer version adaptation step.

Description

The function `ippsFIRLMSOne_Direct` is declared in the `ipps.h` file. This function performs directly a single iteration of FIR filter taps adaptation. The `tapsLen`-length array `pTapsInv` contains the FIR filter taps in the inverse order. The $2 * \text{tapsLen}$ -length array `pDlyLine` specifies the delay line values. The `pDlyLineIndex` array specifies the current index of the delay line. The output signal is stored in `pDstVal`.



NOTE. The adaptation error value can be computed as follows:
 $\text{err}[n] = \text{refVal}[n] - *pDstVal.$

The function `ippsFIRLMSOne_Direct` performs a single iteration of FIR filter taps adaptation with the `mu` step value. The taps are floating-point numbers.

Set the taps to zero or to values close to calculated to speed up the process.

The function `ippsLMSOne_Direct` is to be called within cycle with the number of iterations equal to the number of input samples. You can decide what data is to be saved as a result of adaptation: either the filtered output signal or the adaptation error.

The function `ippsFIRLMSOne_DirectQ15` performs a single iteration of FIR filter taps adaptation with the `muQ15` step value. The taps are integer numbers. The adaptation step `muQ15` can be computed as follows:

$$\text{muQ15} = (\text{int})(\text{mu} * (1 << 15) + 0.5f)$$

[Example 6-6](#) illustrates the use of the function `ippsFIRLMSOne_Direct` to adapt the FIR filter taps. After adaptation the taps are close to 1.0.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>tapsLen</code> is less or equal to 0.

Example 6-6 Usage of the `ippsLMSOne_Direct` Function

```
IppStatus firlmsone(void) {
#define LEN 200
    IppStatus st = ippStsNoErr;
    Ipp32f taps = 0, dly[2] = {0};
    Ipp32f x[LEN], y[LEN], mu = 0.05f;
    int i, indx = 0;
    /// make a const signal of amplitude 1 and noise it
    for( i=0; i<LEN; ++i ) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
    for( i=0; i<LEN-1 && ippStsNoErr==st; ++i )
        st=ippsFIRLMSOne_Direct_32f( x[i], x[i+1], y+1+i, &taps, 1, mu,
            dly, &indx );
    printf_32f("FIRLMSOne tap adapted =", &taps, 1, st );
    return st;
}
```

Output:

```
FIRLMSOne tap adapted = 0.993872
```

Multi-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a multi-rate FIR LMS filter
- get and set the delay line values
- get and set the filter coefficients (taps) values
- set the adaptation step value
- perform filtering
- update the filter coefficients using the result of the filter operation
- free dynamic memory allocated for the functions state

To use the multi-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call [ippsFIRLMSMRInitAlloc](#) to initialize a multi-rate FIR LMS filter.
2. Call [ippsFIRLMSMRPutVal](#) a required number of times to place the input values in the delay line.
3. Call [ippsFIRLMSMROne](#) to filter the samples in the delay line.
4. Call [ippsFIRLMSMRUpdateTaps](#) to update the taps using the value of adaptation error that is based on comparison between filtered and reference signals.
5. Call [ippsFIRLMSMRGetTaps](#) and [ippsFIRLMSMRSetTaps](#) to get and set the filter coefficients (taps). Call [ippsFIRLMSMRGetDlyLine](#) and [ippsFIRLMSMRSetDlyLine](#) to get and set the values in the delay line.
6. Call [ippsFIRLMSMRFree](#) to release dynamic memory associated with the FIR LMS filter.

FIRLMSMRInitAlloc

Allocates memory and initializes an adaptive multi-rate FIR filter that uses the least mean squares (LMS) algorithm.

Syntax

```
IppStatus ippsFIRLMSMRInitAlloc32s_16s(IppsFIRLMSMRState32s_16s** ppState,
    const Ipp32s* pTaps, int tapsLen, const Ipp16s* pDlyLine, int dlyLineIndex,
    int dlyStep, int updateDly, int mu);

IppStatus ippsFIRLMSMRInitAlloc32sc_16sc(IppsFIRLMSMRState32sc_16sc** ppState,
    const Ipp32sc* pTaps, int tapsLen, const Ipp16sc* pDlyLine, int
    dlyLineIndex, int dlyStep, int updateDly, int mu);
```

Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>pDlyLine</i>	Pointer to the array holding the delay line values. The number of elements in the array is <i>tapsLen*dlyStep + updateDly</i> .
<i>dlyLineIndex</i>	Current index of the delay line.

<i>dlyStep</i>	Multi-rate down factor applied to delay line values.
<i>updateDly</i>	Value of adaptation delay in samples.
<i>mu</i>	Adaptation step.
<i>ppState</i>	Pointer to the pointer to the filter state structure.

Description

The function `ippsFIRLMSMRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a multi-rate FIR LMS filter state. The function copies the filter coefficients from *tapsLen*-length array *pTaps* into the state structure *pState*. The array *pDlyLine* specifies the delay line values. The structure is initialized by the downsampling factor over the delay line *dlyStep*, by the adaptation delay value *updateDly*, and by the adaptation step value *mu*. The function returns the pointer in the output parameter *ppState* and also the operation status value.

The function uses copies of the tap values during the filtering procedure. The initial values passed to the function may be obtained from the previous filtering procedure. If the pointer *pTaps* is NULL, then the filter coefficient values are set to zero.

Copies of the input samples are used in the filtering procedure. Values in the delay line represent data in the same format as the input data to be filtered. If the pointer *pDlyLine* is NULL, then the filter delay line values are set to zero.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRFree

Closes an adaptive multi-rate FIR filter that uses the least mean squares algorithm.

Syntax

```

IppStatus ippsFIRLMSMRFree32s_16s(IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRFree32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState);

```


Parameters

pState Pointer to the multi-rate FIR LMS filter state structure to be closed.

Description

The function `ippsFIRLMSMRFree` is declared in the `ipps.h` file. This function closes the multi-rate FIR LMS filter state by freeing all memory associated with a filter state created by [ippsFIRLMSMRInitAlloc](#). Call `ippsFIRLMSMRFree` after filtering is completed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRSetMu

Sets the adaptation step.

Syntax

```
IppStatus ippsFIRLMSMRSetMu32s_16s(IppsFIRLMSMRState32s_16s* pState, const int mu);  
IppStatus ippsFIRLMSMRSetMu32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState, const int mu);
```

Parameters

<i>mu</i>	New adaptation step
<i>pState</i>	Pointer to the filter state structure.

Description

The function `ippsFIRLMSMRSetMu` is declared in the `ipps.h` file. This function updates the adaptation step stored in *pState* with the new value *mu*.

Before calling the function `ippsFIRLMSMRSetMu` the filter state structure must be initialized by the function [ippsFIRLMSMRInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRUpdateTaps

Updates the filter coefficients using the adaptation error value.

Syntax

```
IppStatus ippSFIRLMSMRUpdateTaps32s_16s(Ipp32s errVal,
    IppsFIRLMSMRState32s_16s* pState);
IppStatus ippSFIRLMSMRUpdateTaps32sc_16sc(Ipp32sc errVal,
    IppsFIRLMSMRState32sc_16sc* pState);
```

Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>errVal</i>	Adaptation error value.

Description

The function `ippSFIRLMSMRUpdateTaps` is declared in the `ipps.h` file. This function updates the filter coefficients that are stored in the filter state structure *pState*. It is assumed that the filter operation was performed and the adaptation error value *errVal* was computed before calling this function. The adaptation error value is computed outside the function as the difference between output and reference signals. Updated filter coefficients are defined as

$$h_{n+1}(i) = h_n(i) + \mu \cdot \text{errVal} \cdot x(n - (i \cdot \text{dlyStep}) - \text{updateDly})$$

where $h_{n+1}(i)$ denotes new taps, $h_n(i)$ denotes initial taps, and μ , *errVal* and *updateDly* are the adaptation step, adaptation error value and adaptation delay, respectively.

Before calling the function `ippSFIRLMSMRUpdateTaps` the filter state structure must be initialized by the function [ippsFIRLMSMRInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pState</code> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetTaps

Retrieves tap values from the multi-rate FIR LMS filter.

Syntax

```
IppStatus ippFIRLMSMRGetTaps32s_16s(IppsFIRLMSMRState32s_16s* pState, Ipp32s*
    pOutTaps);
IppStatus ippFIRLMSMRGetTaps32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    Ipp32sc* pOutTaps);
```

Parameters

<code>pState</code>	Pointer to the filter state structure.
<code>pOutTaps</code>	Pointer to the array holding copies of the taps.

Description

The function `ippFIRLMSMRGetTaps` is declared in the `ipps.h` file. This function copies the values of the filter coefficients stored in the filter state structure `pState` to the array pointed by the `pOutTaps` pointer.

Before calling the function `ippFIRLMSMRGetTaps` the filter state structure must be initialized by the function [`ippFIRLMSMRInitAlloc`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRSetTaps

Sets tap values in the multi-rate FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSMRSetTaps32s_16s(IppsFIRLMSMRState32s_16s* pState, const
    Ipp32s* pInTaps);
IppStatus ippsFIRLMSMRSetTaps32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    const Ipp32sc* pInTaps);
```

Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pInTaps</i>	Pointer to the array holding new tap values.

Description

The function `ippsFIRLMSMRSetTaps` is declared in the `ipps.h` file. This function sets the filter coefficients stored in the filter state structure *pState* to the new values stored in an array pointed by the *pInTaps* pointer. If the pointer is `NULL`, then the filter coefficients values are set to zero.

Before calling the function `ippsFIRLMSMRSetTaps` the filter state structure must be initialized by the function [`ippsFIRLMSMRInitAlloc`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetTapsPointer

Returns the pointer to the filter coefficients.

Syntax

```
IppStatus ippsFIRLMSMRGetTapsPointer32s_16s(IppsFIRLMSMRState32s_16s* pState,
      Ipp32s** pTaps);
IppStatusippsFIRLMSMRGetTapsPointer32sc_16sc(IppsFIRLMSMRState32sc_16sc*pState,
      Ipp32sc** pTaps);
```

Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pTaps</i>	Pointer to the variable that contains the pointer to the tap values.

Description

The function `ippsFIRLMSMRGetTapsPointer` is declared in the `ipps.h` file. This function writes the pointer to the filter coefficients stored in the filter state structure *pState* to the variable pointed by *pTaps*.



CAUTION. To get the pointer to tap values directly is faster than to copy them using the `ippsFIRLMSMRGetTaps` function, but this operation may be error-prone.

Before calling the function `ippsFIRLMSMRGetTapsPointer` the filter state structure must be initialized by the function [`ippsFIRLMSMRInitAlloc`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetDlyLine

Retrieves the delay line contents from the multi-rate FIR LMS filter state.

Syntax

```
IppStatus ippsFIRLMSMRGetDlyLine32s_16s(IppsFIRLMSMRState32s_16s* pState,
    Ipp16s* pOutDlyLine, int* pOutDlyLineIndex);
IppStatus ippsFIRLMSMRGetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    Ipp16sc* pOutDlyLine, int* pOutDlyLineIndex);
```

Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pOutDlyLine</i>	Pointer to the array containing the copies of delay line values. The number of elements in the array is <i>tapsLen*dlyStep+updateDly</i> .
<i>pOutDlyLineIndex</i>	Pointer to the array to store the current delay line index copied from <i>pState</i> .

Description

The function `ippsFIRLMSMRGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pOutDlyLine* and *pOutDlyLineIndex*, respectively.

Before calling the function `ippsFIRLMSMRGetDlyLine` the filter state structure must be initialized by the function [ippsFIRLMSMRInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRSetDlyLine

Gets and sets the delay line contents of a multi-rate FIR LMS filter state.

Syntax

```
IppStatus ippsFIRLMSMRSetDlyLine32s_16s(IppsFIRLMSMRState32s_16s* pState, const
    Ipp16s* pInDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSMRSetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    const Ipp16sc* pInDlyLine, int dlyLineIndex);
```

Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pInDlyLine</i>	Pointer to the array containing the new delay line values. The number of elements in the array is <i>tapsLen*dlyStep+updateDly</i> .
<i>dlyLineIndex</i>	Initial index of the delay line to be stored in <i>pState</i> .

Description

The function `ippsFIRLMSMRSetDlyLine` is declared in the `ipps.h` file. This function copies the new values stored in the *pInDlyLine* and corresponding delay line index from *dlyLineIndex* and stores them into the state structure *pState*. If the pointer *pInDlyLine* is `NULL`, the delay line values are set to zero.

Before calling the function `ippsFIRLMSMRSetDlyLine` the filter state structure must be initialized by the function [ippsFIRLMSMRInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetDlyVal

Gets one delay line values from the specified position.

Syntax

```
IppStatus ippsFIRLMSMRGetDlyVal32s_16s(IppsFIRLMSMRState32s_16s* pState,
    Ipp16s* pOutVal, int index);
IppStatus ippsFIRLMSMRGetDlyVal32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    Ipp16sc* pOutVal, int index);
```

Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pOutVal</i>	Pointer to the copied delay line value.
<i>index</i>	Index of the required delay line value.

Description

The function `ippsFIRLMSMRGetDlyVal` is declared in the `ipps.h` file. This function copies from the filter state structure *pState* one value to the *pOutVal*. The position of this sample in the delay line is specified by *index* (which means that *index* iterations ago the sample was placed into the delay line).

Before calling the function `ippsFIRLMSMRGetDlyVal` the filter state structure must be initialized by the function [`ippsFIRLMSMRInitAlloc`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMRPutVal

Places the input value in the delay line.

Syntax

```
IppStatus ippsFIRLMSMRPutVal32s_16s(Ipp16s val, IppsFIRLMSMRState32s_16s*  
    pState);  
IppStatus ippsFIRLMSMRPutVal32sc_16sc(Ipp16sc val, IppsFIRLMSMRState32sc_16sc*  
    pState);
```

Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>val</i>	Value of the input sample.

Description

The function `ippsFIRLMSMRPutVal` is declared in the `ipps.h` file. This function places the value of the input sample *val* into the delay line, thus preparing the filter with the state structure *pState* for the filtering procedure.

Before calling the function `ippsFIRLMSMRPutVal` the filter state structure must be initialized by the function [`ippsFIRLMSMRInitAlloc`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMROne

Filters data placed in the delay line.

Syntax

```
IppStatus ippsFIRLMSMROne32s_16s(Ipp32s* pDstVal, IppsFIRLMSMRState32s_16s*
    pState);
IppStatus ippsFIRLMSMROne32sc_16sc(Ipp32sc* pDstVal, IppsFIRLMSMRState32sc_16sc*
    pState);
```

Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pDstVal</i>	Pointer to the output signal value.

Description

The function `FIRLMSMROne` is declared in the `ipps.h` file. This function filters the samples placed in the delay line using the filter coefficients stored in the filter state structure *pState*. The resulting value is placed in the *pDstVal*. The downsampling factor *dlyStep* defines the number of samples that are filtered. The filter coefficients are not updated.

The filtering procedure can be described as a FIR filter operation (here the input sample to be filtered is denoted $x(n)$, the taps are denoted $h(i)$, and the return value is $y(n)$):

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n - (i \cdot dlyStep))$$

Note that the function operates with values stored in the delay line that are copies of the input samples.

Before calling the function `ippsFIRLMSMROne`, initialize the filter state by calling [`ippsFIRLMSMRInitAlloc`](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

FIRLMSMROneVal

Filters one input value.

Syntax

```
IppStatus ippSFIRLMSMROneVal32s_16s(Ipp16s val, Ipp32s* pDstVal,  
    IppsFIRLMSMRState32s_16s* pState);  
  
IppStatus ippSFIRLMSMROneVal32sc_16sc(Ipp16sc val, Ipp32sc* pDstVal,  
    IppsFIRLMSMRState32sc_16sc* pState);
```

Parameters

<code>pState</code>	Pointer to the filter state structure.
<code>pDstVal</code>	Pointer to the output signal value.
<code>val</code>	Value of the input signal sample.

Description

The function `ippSFIRLMSMROneVal` is declared in the `ipps.h` file. This function places one input sample `val` into the delay line and filters it using the filter coefficients specified in the filter state structure `pState`. The result value is stored in the `pDstVal`.

Before calling the function `ippSFIRLMSMROneVal`, initialize the filter state by calling [`ippsFIRLMSMRInitAlloc`](#).

Return Values

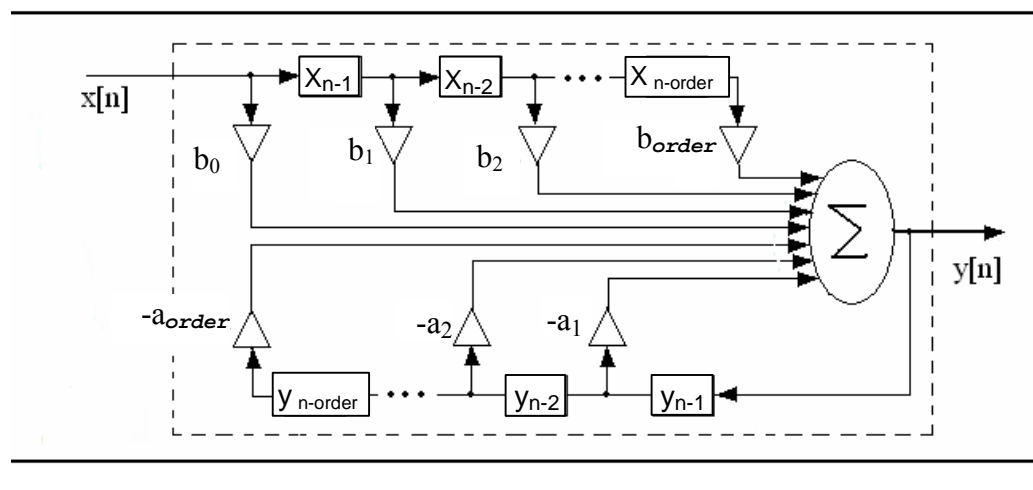
<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

IIR Filter Functions

The functions described in this section initialize an infinite impulse response (IIR) filter and perform filtering. Intel IPP supports two types of filters: arbitrary order filter and biquad filter.

[Figure 6-1](#) shows the structure of an arbitrary order IIR filter.

Figure 6-1 Structure of an Arbitrary Order Filter



Here $x[n]$ is a sample of the input signal, $y[n]$ is a sample of the output signal, $order$ is the filter order, and $b_0, b_1, \dots, b_{order}, a_1, \dots, a_{order}$ are the reduced filter coefficients:

The output signal is computed by the following formula:

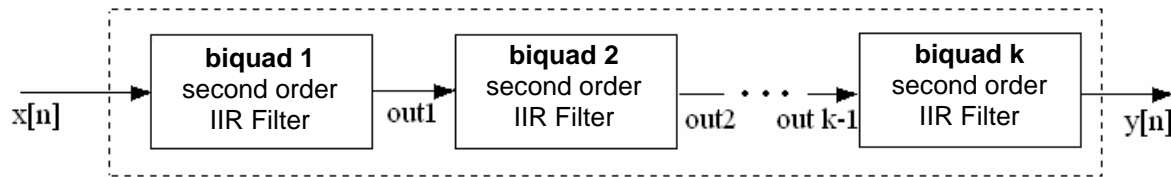
$$y[n] = \sum_{k=0}^{order} b_k \cdot x(n-k) - \sum_{k=1}^{order} a_k \cdot y(n-k)$$

Reduced coefficients are calculated as $a_k = \frac{A_k}{A_0}$ and $b_k = \frac{B_k}{A_0}$

where $A_0, A_1, \dots, A_{order}, B_0, B_1, \dots, B_{order}$ are initial filter coefficients (taps).

A biquad IIR filter is a cascade of second-order filters. The [Figure 6-2](#) illustrates the structure of the biquad filter with k cascades of second-order filters.

Figure 6-2 **Structure of a BiQuad IIR Filter**



To initialize and use an IIR filter, follow this general scheme:

1. Call [ippsIIRInitAlloc](#) to allocate memory and initialize the filter as an arbitrary order IIR filter, or call [ippsIIRInitAlloc_BiQuad](#) to allocate memory and initialize the filter as a cascade of biquads.
Or alternatively call either [ippsIIRInit](#) to initialize the filter as an arbitrary order IIR filter in the external buffer, or [ippsIIRInit_BiQuad](#) to initialize the filter as a cascade of biquads in the external buffer. Size of the buffer can be computed by calling the functions [ippsIIRGetStateSize](#) or [ippsIIRGetStateSize_BiQuad](#), respectively.
2. Call [ippsIIROne](#) repeatedly to filter a single sample through an IIR filter or call [ippsIIR](#) to filter consecutive samples at once.
3. Call [ippsIIRGetDlyLine](#) and [ippsIIRSetDlyLine](#) to get and set the delay line values in the IIR state structure.
4. Call [ippsIIRSetTaps](#) to set new tap values in the previously initialized filter state structure.
5. After all filtering is complete, call [ippsIIRFree](#) to release dynamic memory associated with the filter state structure created by [ippsIIRInitAlloc](#) or [ippsIIRInitAlloc_BiQuad](#).

Alternatively, you may use the direct version of the functions. These functions perform filtering without initializing the filter state structure. All required parameters are directly set in the function.

IIRInitAlloc

Allocates memory and initializes the state structure for an arbitrary IIR filter.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsIIRInitAlloc32s_16s(IppsIIRState32s_16s** ppState, const Ipp32s*
    pTaps, int order, int tapsFactor, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32s_16s32f(IppsIIRState32s_16s** ppState, const
    Ipp32f* pTaps, int order, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32f_16s(IppsIIRState32f_16s** ppState, const Ipp32f*
    pTaps, int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_16s(IppsIIRState64f_16s** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32s(IppsIIRState64f_32s** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc32sc_16sc(IppsIIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int order, int tapsFactor, const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_16sc32fc(IppsIIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32fc_16sc(IppsIIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_16sc(IppsIIRState64fc_16sc** ppState, const
    Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_32sc(IppsIIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsIIRInitAlloc_32f(IppsIIRState_32f** ppState, const Ipp32f* pTaps,
    int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32f(IppsIIRState64f_32f** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine);
```

```

IppStatus ippsIIRInitAlloc_64f(IppsIIRState_64f** ppState, const Ipp64f* pTaps,
    int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc_32fc(IppsIIRState_32fc** ppState, const Ipp32fc*
    pTaps, int order, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_32fc(IppsIIRState64fc_32fc** ppState, const
    Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc_64fc(IppsIIRState_64fc** ppState, const Ipp64fc*
    pTaps, int order, const Ipp64fc* pDlyLine);

```

Parameters

<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $2*(order+1)$.
<i>tapsFactor</i>	Scale factor for the taps of integer data type.
<i>order</i>	Order of the IIR filter.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>order</i> .
<i>ppState</i>	Pointer to the pointer to the IIR state structure.

Description

The function `ippsIIRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes an arbitrary IIR filter state. The initialization functions copy the taps from the array *pTaps* into the state structure *ppState*. The *tapsFactor* is used to scale integer tap values. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0. The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The $2*(order + 1)$ -length array *pTaps* specifies the taps arranged in the array as follows:

$$B_0, B_1, \dots, B_{order}, A_0, A_1, \dots, A_{order}$$

$$A_0 \neq 0$$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. [Example 6-8](#) shows how to convert floating-point taps into integer data type.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>order</i> is less than 0.
<code>ippStsDivByZeroErr</code>	Indicates an error when A_0 is equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

IIRInitAlloc_BiQuad

Allocates memory and initializes the state structure for the biquad IIR filter.

Syntax

Case 1: Operation on integer samples.

```

IppStatus ippSIIRInitAlloc32s_BiQuad_16s(IppsIIRState32s_16s** ppState, const
    Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine);
IppStatus ippSIIRInitAlloc32s_BiQuad_16s32f(IppsIIRState32s_16s** ppState,
    const Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine);
IppStatus ippSIIRInitAlloc32f_BiQuad_16s(IppsIIRState32f_16s** ppState, const
    Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);
IppStatus ippSIIRInitAlloc64f_BiQuad_16s(IppsIIRState64f_16s** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippSIIRInitAlloc64f_BiQuad_32s(IppsIIRState64f_32s** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);

IppStatus ippSIIRInitAlloc32sc_BiQuad_16sc(IppsIIRState32sc_16sc** ppState,
    const Ipp32sc* pTaps, int numBq, int tapsFactor, const Ipp32sc* pDlyLine);
IppStatus ippSIIRInitAlloc32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc** ppState,
    const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine);
IppStatus ippSIIRInitAlloc32fc_BiQuad_16sc(IppsIIRState32fc_16sc** ppState,
    const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);

```



```
IppStatus ippsIIRInitAlloc64fc_BiQuad_16sc(IppsIIRState64fc_16sc** ppState,
    const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_32sc(IppsIIRState64fc_32sc** ppState,
    const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsIIRInitAlloc_BiQuad_32f(IppsIIRState_32f** ppState, const Ipp32f*
    pTaps, int numBq, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_BiQuad_32f(IppsIIRState64f_32f** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_64f(IppsIIRState_64f** ppState, const Ipp64f*
    pTaps, int numBq, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc_BiQuad_32fc(IppsIIRState_32fc** ppState, const
    Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_32fc(IppsIIRState64fc_32fc** ppState,
    const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_64fc(IppsIIRState_64fc** ppState, const
    Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

Parameters

<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $6 * numBq$.
<i>tapsFactor</i>	Scale factor for the taps of integer data type.
<i>numBq</i>	Number of cascades of biquads..
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * numBq$.
<i>ppState</i>	Pointer to the pointer to the biquad IIR state structure.

Description

The function `ippsIIRInitAlloc_BiQuad` is declared in the `ipps.h` file. This function allocates memory and initializes a biquad (BQ) IIR filter state. The initialization functions copy the taps from the array *pTaps* into the state structure *ppState*. The *tapsFactor* is used to scale integer tap values. The $2 * numBq$ -length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

A biquad IIR filter is defined by a cascade of biquads. The number of cascades of biquads is specified by the *numBq* value. The $6 * numBq$ -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \dots A_{numBq-1,2}$

$A_{n,0} \neq 0, B_{n,0} \neq 0$

If the state is not created, the initialization function returns an error status.

The initialization functions with the *32s_32f* suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. [Example 6-8](#) shows how to convert floating-point taps into integer data type.

Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsMemAllocErr</i>	Indicates an error when no memory allocated.
<i>ippStsNullPtrErr</i>	Indicates an error when one of the specified pointers is NULL.
<i>ippStsIIROrderErr</i>	Indicates an error when <i>numBq</i> is less than 1.
<i>ippStsDivByZeroErr</i>	Indicates an error when A_0 , $A_{n,0}$ or $B_{n,0}$ is equal to 0.
<i>ippStsContextMatchErr</i>	Indicates an error when the state identifier is incorrect.

IIRFree

Closes an IIR filter state.

Syntax

```

IppStatus ippIIRFree_32f(IppsIIRState_32f* pState);
IppStatus ippIIRFree_64f(IppsIIRState_64f* pState);
IppStatus ippIIRFree_32fc(IppsIIRState_32fc* pState);
IppStatus ippIIRFree_64fc(IppsIIRState_64fc* pState);
IppStatus ippIIRFree32s_16s(IppsIIRState32s_16s* pState);
IppStatus ippIIRFree32sc_16sc(IppsIIRState32sc_16sc* pState);

```

```

IppStatus ippSIIRFree32f_16s(IppsIIRState32f_16s* pState);
IppStatus ippSIIRFree32fc_16sc(IppsIIRState32fc_16sc* pState);
IppStatus ippSIIRFree64f_16s(IppsIIRState64f_16s* pState);
IppStatus ippSIIRFree64f_32s(IppsIIRState64f_32s* pState);
IppStatus ippSIIRFree64f_32f(IppsIIRState64f_32f* pState);
IppStatus ippSIIRFree64fc_16sc(IppsIIRState64fc_16sc* pState);
IppStatus ippSIIRFree64fc_32sc(IppsIIRState64fc_32sc* pState);
IppStatus ippSIIRFree64fc_32fc(IppsIIRState64fc_32fc* pState);

```

Parameters

pState Pointer to an IIR filter state structure to be closed.

Description

The function `ippSIIRFree` is declared in the `ipps.h` file. This function closes the IIR filter state by freeing all memory associated with a filter state created by [ippSIIRInitAlloc](#) or [ippSIIRInitAlloc_BiQuad](#). Call `ippSIIRFree` after filtering is completed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

IIRInit

Initializes an arbitrary IIR filter state.

Syntax

Case 1: Operation on integer samples.

```

IppStatus ippSIIRInit32s_16s(IppsIIRState32s_16s** ppState, const Ipp32s*
    pTaps, int order, int tapsFactor, const Ipp32s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippSIIRInit32s_16s32f(IppsIIRState32s_16s** ppState, const Ipp32f*
    pTaps, int order, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

```

```

IppStatus ippsIIRInit32f_16s(IppsIIRState32f_16s** ppState, const Ipp32f*
    pTaps, int order, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_16s(IppsIIRState64f_16s** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_32s(IppsIIRState64f_32s** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32sc_16sc(IppsIIRState32sc_16sc** ppState, const Ipp32sc*
    pTaps, int order, int tapsFactor, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32sc_16sc32fc(IppsIIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32fc_16sc(IppsIIRState32fc_16sc** ppState, const Ipp32fc*
    pTaps, int order, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_16sc(IppsIIRState64fc_16sc** ppState, const Ipp64fc*
    pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_32sc(IppsIIRState64fc_32sc** ppState, const Ipp64fc*
    pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

```

Case 2: Operation on floating point samples.

```

IppStatus ippsIIRInit_32f(IppsIIRState_32f** ppState, const Ipp32f* pTaps, int
    order, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_32f(IppsIIRState64f_32f** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_64f(IppsIIRState_64f** ppState, const Ipp64f* pTaps, int
    order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_32fc(IppsIIRState_32fc** ppState, const Ipp32fc* pTaps,
    int order, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_32fc(IppsIIRState64fc_32fc** ppState, const Ipp64fc*
    pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_64fc(IppsIIRState_64fc** ppState, const Ipp64fc* pTaps,
    int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

```

Parameters

<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $2 \times (\text{order} + 1)$.
<i>tapsFactor</i>	Scale factor for the taps of integer data type.

<i>order</i>	Order of the IIR filter.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>order</i> .
<i>ppState</i>	Pointer to the pointer to the arbitrary IIR state structure to be created.
<i>pBuffer</i>	Pointer to the external buffer.

Description

The function `ippsIIRInit` is declared in the `ipps.h` file. This function initializes an arbitrary IIR filter state in the external buffer. The size of this buffer should be computed previously by calling the function [IIRGetStateSize](#). The initialization functions copy the taps from the array *pTaps* into the state structure *pState*. The *tapsFactor* is used to scale integer tap values. The *order*-length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The $2 \cdot (\text{order} + 1)$ -length array *pTaps* specifies the taps arranged in the array as follows:

$$B_0, B_1, \dots, B_{\text{order}}, A_0, A_1, \dots, A_{\text{order}}$$

$$A_0 \neq 0$$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. [Example 6-8](#) shows how to convert floating-point taps into integer data type.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippsStsDivByZeroErr</code>	Indicates an error when A_0 is equal to 0.
<code>ippsStsIIROrderErr</code>	Indicates an error when <i>order</i> is less than 0.

IIRInit_BiQuad

Initializes an IIR filter state.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsIIRInit32s_BiQuad_16s(IppsIIRState32s_16s** ppState, const
    Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine, Ipp8u*
    pBuffer);

IppStatus ippsIIRInit32s_BiQuad_16s32f(IppsIIRState32s_16s** ppState, const
    Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32f_BiQuad_16s(IppsIIRState32f_16s** ppState, const
    Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64f_BiQuad_16s(IppsIIRState64f_16s** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64f_BiQuad_32s(IppsIIRState64f_32s** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32sc_BiQuad_16sc(IppsIIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int numBq, int tapsFactor, const Ipp32sc* pDlyLine, Ipp8u*
    pBuffer);

IppStatus ippsIIRInit32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc** ppState,
    const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32fc_BiQuad_16sc(IppsIIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_BiQuad_16sc(IppsIIRState64fc_16sc** ppState, const
    Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_BiQuad_32sc(IppsIIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsIIRInit_BiQuad_32f(IppsIIRState_32f** ppState, const Ipp32f*
    pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64f_BiQuad_32f(IppsIIRState64f_32f** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
```

```

IppStatus ippsIIRInit_BiQuad_64f(IppsIIRState_64f** ppState, const Ipp64f*
    pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_BiQuad_32fc(IppsIIRState_32fc** ppState, const Ipp32fc*
    pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_BiQuad_32fc(IppsIIRState64fc_32fc** ppState, const
    Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_BiQuad_64fc(IppsIIRState_64fc** ppState, const Ipp64fc*
    pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

```

Parameters

<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $6 * numBq$.
<i>tapsFactor</i>	Scale factor for the taps of integer data type.
<i>numBq</i>	Number of cascades of biquads..
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * numBq$.
<i>ppState</i>	Pointer to the pointer to the biquad IIR state structure.
<i>pBuffer</i>	Pointer to the external buffer .

Description

The function `ippsIIRInit_BiQuad` are declared in the `ipps.h` file. This function initializes a biquad (BQ) IIR filter state in the external buffer. The size of this buffer should be computed previously by calling the function [IIRGetStateSize_BiQuad](#). The initialization function copies the taps from the array *pTaps* into the state structure *ppState*. The *tapsFactor* is used to scale integer tap values. The $2 * numBq$ -length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

A biquad IIR filter is defined by a cascade of biquads. The number of cascades of biquads is specified by the *numBq* value. The $6 * numBq$ -length array *pTaps* specifies the taps arranged in the array as follows:

$$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \dots A_{numBq-1,2}$$

$$A_{n,0} \neq 0, B_{n,0} \neq 0$$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. [Example 6-8](#) shows how to convert floating-point taps into integer data type.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsDivByZeroErr</code>	Indicates an error when A_0 , $A_{n,0}$ or $B_{n,0}$ is equal to 0.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>numBq</i> is less than 1.

IIRGetStateSize

Computes the length of the external buffer for the arbitrary IIR filter state structure.

Syntax

```

IppStatus ippSIIRGetStateSize32s_16s(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize32s_16s32f(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize32f_16s(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize64f_16s(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize64f_32s(int order, int* pBufferSize);

IppStatus ippSIIRGetStateSize32sc_16sc(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize32sc_16sc32fc(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize32fc_16sc(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize64fc_16sc(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize64fc_32sc(int order, int* pBufferSize);

IppStatus ippSIIRGetStateSize_32f(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize64f_32f(int order, int* pBufferSize);
IppStatus ippSIIRGetStateSize_64f(int order, int* pBufferSize);

```



```
IppStatus ippsIIRGetStateSize_32fc(int order, int* pBufferSize);  
IppStatus ippsIIRGetStateSize64fc_32fc(int order, int* pBufferSize);  
IppStatus ippsIIRGetStateSize_64fc(int order, int* pBufferSize);
```

Parameters

<i>order</i>	Order of the IIR filter.
<i>pBufferSize</i>	Pointer to the computed buffer size value.

Description

The function `ippsIIRGetStateSize` is declared in the `ipps.h` file. This function computes the size of the external buffer for an arbitrary IIR filter state, and stores the result in *pBufferSize*.

To compute a size of the buffer, the filter order argument *order* should be specified.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pStateSize</i> pointer is NULL.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>order</i> is less than 0.

IIRGetStateSize_BiQuad

Computes the length of the external buffer for the biquad IIR filter state structure.

Syntax

```
IppStatus ippsIIRGetStateSize32s_BiQuad_16s(int numBq, int* pBufferSize);  
IppStatus ippsIIRGetStateSize32s_BiQuad_16s32f(int numBq, int* pBufferSize);  
IppStatus ippsIIRGetStateSize32f_BiQuad_16s(int numBq, int* pBufferSize);  
IppStatus ippsIIRGetStateSize64f_BiQuad_16s(int numBq, int* pBufferSize);  
IppStatus ippsIIRGetStateSize64f_BiQuad_32s(int numBq, int* pBufferSize);  
  
IppStatus ippsIIRGetStateSize32sc_BiQuad_16sc(int numBq, int* pBufferSize);
```

```

IppStatus ippsIIRGetStateSize32sc_BiQuad_16sc32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize32fc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_32sc(int numBq, int* pBufferSize);

IppStatus ippsIIRGetStateSize_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64f(int numBq, int* pBufferSize);

IppStatus ippsIIRGetStateSize_BiQuad_32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64fc(int order, int* pBufferSize);

```

Parameters

<i>numBq</i>	Number of cascades of biquads.
<i>pBufferSize</i>	Pointer to the computed buffer size value.

Description

The function `ippsIIRGetStateSize_BiQuad` is declared in the `ipps.h` file. This function computes the size of the external buffer for a biquad IIR filter state, and store the result in *pBufferSize*.

To compute a size of the buffer, the number of cascades of biquads *numBq* should be specified.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pStateSize</i> pointer is NULL.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>numBq</i> is less than 1.

IIRSetTaps

Sets the taps in an IIR filter state.

Syntax

```

IppStatus ippsIIRSetTaps_32f(const Ipp32f* pTaps, IppsIIRState_32f* pState);
IppStatus ippsIIRSetTaps_32fc(const Ipp32fc* pTaps, IppsIIRState_32fc* pState);
IppStatus ippsIIRSetTaps_64f(const Ipp64f* pTaps, IppsIIRState_64f* pState);
IppStatus ippsIIRSetTaps_64fc(const Ipp64fc* pTaps, IppsIIRState_64fc* pState);
IppStatus ippsIIRSetTaps32s_16s(const Ipp32s* pTaps, IppsIIRState32s_16s*
    pState, int tapsFactor);
IppStatus ippsIIRSetTaps32s_16s32f(const Ipp32f* pTaps, IppsIIRState32s_16s*
    pState);
IppStatus ippsIIRSetTaps32sc_16sc(const Ipp32sc* pTaps, IppsIIRState32sc_16sc*
    pState, int tapsFactor);
IppStatus ippsIIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps,
    IppsIIRState32sc_16sc* pState);
IppStatus ippsIIRSetTaps32f_16s(const Ipp32f* pTaps, IppsIIRState32f_16s*
    pState);
IppStatus ippsIIRSetTaps32fc_16sc(const Ipp32fc* pTaps, IppsIIRState32fc_16sc*
    pState);
IppStatus ippsIIRSetTaps64f_16s(const Ipp64f* pTaps, IppsIIRState64f_16s*
    pState);
IppStatus ippsIIRSetTaps64f_32s(const Ipp64f* pTaps, IppsIIRState64f_32s*
    pState);
IppStatus ippsIIRSetTaps64f_32f(const Ipp64f* pTaps, IppsIIRState64f_32f*
    pState);
IppStatus ippsIIRSetTaps64fc_16sc(const Ipp64fc* pTaps, IppsIIRState64fc_16sc*
    pState);
IppStatus ippsIIRSetTaps64fc_32sc(const Ipp64fc* pTaps, IppsIIRState64fc_32sc*
    pState);
IppStatus ippsIIRSetTaps64fc_32fc(const Ipp64fc* pTaps, IppsIIRState64fc_32fc*
    pState);

```

Parameters

<i>pTaps</i>	Pointer to the array containing the tap values.
<i>pState</i>	Pointer to the IIR filter state structure.
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer versions only).

Description

The function `ippsIIRSetTaps` is declared in the `ipps.h` file. This function sets new tap values in the previously initialized IIR filter state structure *pState*. New tap values must be specified in the array *pTaps*. To scale integer taps use the *tapsFactor* value.

The filter state must be initialized before calling the function `ippsIIRSetTaps`. The length of the array *pTaps* should be equal to the *tapsLen* parameter of the initialized filter state.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsDivByZeroErr</code>	Indicates an error when when A_0 is equal to 0 for an arbitrary IIR filter state, or A_0 , $A_{n,0}$ or $B_{n,0}$ is equal to 0 for a biquad IIR filter state.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

IIRGetDlyLine

Retrieves the delay line contents from the IIR filter state.

Syntax

```
IppStatus ippsIIRGetDlyLine32s_16s(IppsIIRState32s_16s* pState, const Ipp32s*
    pDlyLine);
IppStatus ippsIIRGetDlyLine32f_16s(IppsIIRState32f_16s* pState, const Ipp32f*
    pDlyLine);
```

```

IppStatus ippsIIRGetDlyLine64f_16s(IppsIIRState64f_16s* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32s(IppsIIRState64f_32s* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine32sc_16sc(IppsIIRState32sc_16sc* pState, const
    Ipp32sc* pDlyLine);
IppStatus ippsIIRGetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState, const
    Ipp32fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState, const
    Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState, const
    Ipp64fc* pDlyLine);

IppStatus ippsIIRGetDlyLine_32f(IppsIIRState_32f* pState, const Ipp32f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32f(IppsIIRState64f_32f* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine_64f(IppsIIRState_64f* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine_32fc(IppsIIRState_32fc* pState, const Ipp32fc*
    pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState, const
    Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine_64fc(IppsIIRState_64fc* pState, const Ipp64fc*
    pDlyLine);

```

Parameters

<i>pState</i>	Pointer to the IIR filter state structure.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>order</i> for arbitrary filters and $2 * numBq$ for BQ filters.

Description

The function `ippsIIRGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from the state structure *pState* and stores them into the arrays *pDlyLine*. If the pointer is NULL, then the delay line values in the state structure are initialized to zero.

The filter state must be previously initialized by the one of the initialization functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

IIRSetDlyLine

Sets the delay line contents in an IIR filter state.

Syntax

```

IppStatus ippSIIRSetDlyLine32s_16s(IppsIIRState32s_16s* pState, const Ipp32s*
    pDlyLine);
IppStatus ippSIIRSetDlyLine32f_16s(IppsIIRState32f_16s* pState, const Ipp32f*
    pDlyLine);
IppStatus ippSIIRSetDlyLine64f_16s(IppsIIRState64f_16s* pState, const Ipp64f*
    pDlyLine);
IppStatus ippSIIRSetDlyLine64f_32s(IppsIIRState64f_32s* pState, const Ipp64f*
    pDlyLine);
IppStatus ippSIIRSetDlyLine32sc_16sc(IppsIIRState32sc_16sc* pState, const
    Ipp32sc* pDlyLine);
IppStatus ippSIIRSetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState, const
    Ipp32fc* pDlyLine);
IppStatus ippSIIRSetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState, const
    Ipp64fc* pDlyLine);
IppStatus ippSIIRSetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState, const
    Ipp64fc* pDlyLine);

IppStatus ippSIIRSetDlyLine_32f(IppsIIRState_32f* pState, const Ipp32f*
    pDlyLine);
IppStatus ippSIIRSetDlyLine64f_32f(IppsIIRState64f_32f* pState, const Ipp64f*
    pDlyLine);
IppStatus ippSIIRSetDlyLine_64f(IppsIIRState_64f* pState, const Ipp64f*
    pDlyLine);

```

```

IppStatus ippsIIRSetDlyLine_32fc(IppsIIRState_32fc* pState, const Ipp32fc*
    pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState, const
    Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine_64fc(IppsIIRState_64fc* pState, const Ipp64fc*
    pDlyLine);

```

Parameters

<i>pState</i>	Pointer to the IIR filter state structure.
<i>pDlyLine</i>	Pointer to the array holding the delay line values. The number of elements in the array is <i>order</i> for arbitrary filters and $2 * numBq$ for BQ filters. If the pointer is NULL, then the delay line values in the state structure are initialized to zero.

Description

The function `ippsIIRSetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from *pDlyLine* and stores them into the state structure *pState*. If the pointer is NULL, then the delay line values in the state structure are initialized to zero.

The filter state must be previously initialized by the one of the initialization functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

IIROne

Filters a single sample through an IIR filter.

Syntax

```

IppStatus ippsIIROne_32f(Ipp32f src, Ipp32f* pDstVal, IppsIIRState_32f*
    pState);
IppStatus ippsIIROne_64f(Ipp64f src, Ipp64f* pDstVal, IppsIIRState_64f*
    pState);

```

```

IppStatus ippsIIROne64f_32f(Ipp32f src, Ipp32f* pDstVal, IppsIIRState64f_32f*
    pState);
IppStatus ippsIIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsIIRState_32fc*
    pState);
IppStatus ippsIIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal, IppsIIRState_64fc*
    pState);
IppStatus ippsIIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsIIRState64fc_32fc*
    pState);

IppStatus ippsIIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsIIRState32s_16s*
    pState, int scaleFactor);
IppStatus ippsIIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsIIRState32f_16s*
    pState, int scaleFactor);
IppStatus ippsIIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsIIRState64f_16s*
    pState, int scaleFactor);
IppStatus ippsIIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, IppsIIRState64f_32s*
    pState, int scaleFactor);

IppStatus ippsIIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
    IppsIIRState64fc_32sc* pState, int scaleFactor);

```

Parameters

<i>pState</i>	Pointer to the IIR filter state structure.
<i>src</i>	Input sample.
<i>pDstVal</i>	Pointer to the output filtered sample.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsIIROne` is declared in the `ipps.h` file. This function filters a single sample `src` through an IIR filter with real taps, and stores the result in `pDstVal`. The filter parameters are specified in `pState`. The output of the integer sample is scaled according to `scaleFactor` and can be saturated.

Do not modify the `scaleFactor` value unless the state structure is changed.

The filter state must be initialized before calling either the `ippsIIR` or `ippsFIROne` function. Specify the number of taps `tapsLen`, the tap values in `pTaps`, the delay line values in `pDlyLine`, and the `order` or `numBq` value beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

IIR

Filters a block of samples through an IIR filter.

Syntax

Case 1: Not-in-place operation on integer samples.

```
IppStatus ippsIIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    IppsIIRState64f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    IppsIIRState64f_32s* pState, int scaleFactor);

IppStatus ippsIIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    IppsIIRState32sc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsIIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int len,
    IppsIIRState64fc_32sc* pState, int scaleFactor);
```

Case 2: Not-in-place operation on floating point samples.

```
IppStatus ippsIIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, IppsIIRState_32f*
    pState);
IppStatus ippsIIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, IppsIIRState_64f*
    pState);
IppStatus ippsIIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    IppsIIRState64f_32f* pState);
IppStatus ippsIIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, IppsIIRState_32fc*
    pState);
IppStatus ippsIIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, IppsIIRState_64fc*
    pState);
IppStatus ippsIIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    IppsIIRState64fc_32fc* pState);
```

Case 3: In-place operation on integer samples.

```
IppStatus ippsIIR32s_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState32s_16s*
    pState, int scaleFactor);
IppStatus ippsIIR32f_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState32f_16s*
    pState, int scaleFactor);
IppStatus ippsIIR64f_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState64f_16s*
    pState, int scaleFactor);
IppStatus ippsIIR64f_32s_ISfs(Ipp32s* pSrcDst, int len, IppsIIRState64f_32s*
    pState, int scaleFactor);

IppStatus ippsIIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int len, IppsIIRState32fc_16sc*
    pState, int scaleFactor);
IppStatus ippsIIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int len, IppsIIRState32sc_16sc*
    pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int len, IppsIIRState64fc_16sc*
    pState, int scaleFactor);
```

```
IppStatus ippsIIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int len, IppsIIRState64fc_32sc*
    pState, int scaleFactor);
```

Case 4: In-place operation on floating point samples.

```
IppStatus ippsIIR_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState_32f* pState);
IppStatus ippsIIR_64f_I(Ipp64f* pSrcDst, int len, IppsIIRState_64f* pState);
IppStatus ippsIIR64f_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState64f_32f*
    pState);
IppStatus ippsIIR_32fc_I(Ipp32fc* pSrcDst, int len, IppsIIRState_32fc* pState);
IppStatus ippsIIR_64fc_I(Ipp64fc* pSrcDst, int len, IppsIIRState_64fc* pState);
IppStatus ippsIIR64fc_32fc_I(Ipp32fc* pSrcDst, int len, IppsIIRState64fc_32fc*
    pState);
```

Parameters

<i>pState</i>	Pointer to the IIR filter state structure.
<i>pSrc</i>	Pointer to the input array of samples.
<i>pDst</i>	Pointer to the output array of filtered samples.
<i>pSrcDst</i>	Pointer to the input and output array of samples for the in-place operations.
<i>len</i>	Number of samples to be filtered.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsIIR` is declared in the `ipps.h` file. This function filters *len* samples in the input array *pSrc* or *pSrcDst* through an IIR filter, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

The filter state must be initialized before calling the `ippsIIR` function. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

[Example 6-7](#) illustrates using `ippsIIR_32f` to suppress a 60 Hz signal.

[Example 6-8](#) illustrates using `ippsIIR` to filter a sample. The function `ippsCnvert_64f32s_Sfs` converts floating-point taps into integer data type before calling `ippsIIRInitAlloc_32s`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.

Example 6-7 Using the `ippsIIR_32f` Function to Suppress a 60 Hz Signal

```

IppStatus iir( void ) {
#undef NUMITERS
#define NUMITERS 150
    int n;
    IppStatus status;
    IppsIIRState_32f *ctx;
    Ipp32f *x = ippsMalloc_32f( NUMITERS ), *y = ippsMalloc_32f( NUMITERS );
    /// A second-order notch filter having notch freq at 60 Hz
    const float taps[] = {
        0.940809f, -1.105987f, 0.940809f, 1, -1.105987f, 0.881618f
    };
    /// generate a signal having 60 Hz freq sampled with 400 Hz freq
    for(n=0; n<NUMITERS; ++n) x[n] = (float) sin(IPP_2PI * n * 60 / 400);
    ippsIIRInitAlloc_32f( &ctx, taps, 2, NULL );
    status = ippsIIR_32f( x, y, NUMITERS, ctx );
    printf_32f( " IIR 32f output+120 =", y+120, 5, status );
    ippsIIRFree_32f( ctx );
    ippsFree( y );
    ippsFree( x );
    return status;
}

```

Output:

IIR 32f output + 120 = -0.000094 0.000339 0.000458 0.000208 -0.000173

Matlab* Analog:

```

>> B = [0.940809, -1.105987, 0.940809]; A = [1, -1.105987, 0.881618];
n = 0:150; x = sin(2*pi*n*60/400); y = filter(B,A,x); y(121:125)

```

Example 6-8 Using the ippsIIR Function to Filter a Sample

```

IppStatus iirl6s( void ) {
#undef NUMITERS
#define NUMITERS 150
    int n, tapsfactor = 30;
    IppStatus status;
    IppsIIRState32s_16s *ctx;
    Ipp16s *x = ippsMalloc_16s( NUMITERS ), *y = ippsMalloc_16s( NUMITERS );
    /// A second-order notch filter having notch freq at 60 Hz
    Ipp64f taps[6] = {
        0.940809f, -1.105987f, 0.940809f, 1, -1.105987f, 0.881618f
    };
    Ipp32s taps32s[6];
    Ipp64f tmax, tmp[6];
    ippsAbs_64f( taps, tmp, 6 );
    ippsMax_64f( tmp, 6, &tmax );
    tapsfactor = 0;
    if( tmax > IPP_MAX_32S )
        while( (tmax/=2) > IPP_MAX_32S ) ++tapsfactor;
    else
        while( (tmax*=2) < IPP_MAX_32S ) --tapsfactor;
    if( tapsfactor > 0 )
        ippsDivC_64f_I( (float)(1<<(++tapsfactor)), taps, 6 );
    else if( tapsfactor < 0 )
        ippsMulC_64f_I( (float)(1<<(-(tapsfactor))), taps, 6 );
    ippsConvert_64f32s_Sfs( taps, taps32s, 6, ipprndNear, 0 );
    /// generate a signal of 60 Hz freq that is sampled with 400 Hz freq
    for(n=0; n<NUMITERS; ++n) x[n] = (Ipp16s)(1000*sin(IPP_2PI*n*60/400));
    ippsIIRInitAlloc32s_16s( &ctx, taps32s, 2, tapsfactor, NULL );
    status = ippsIIR32s_16s_Sfs( x, y, NUMITERS, ctx, 0 );
    printf_16s( " IIR 32s output+120 =", y+120, 5, status );
    ippsIIRFree32s_16s( ctx );
    ippsFree( y );
    ippsFree( x );
    return status;
}

```

Output:

```
IIR 32s output + 120 =  0 0 0 0 0
```

IIROne_Direct

Directly filters a single sample through the arbitrary IIR filter.

Syntax

```
IppStatus ippsIIROne_Direct_16s(Ipp16s src, Ipp16s* pDstVal, const Ipp16s*
    pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippsIIROne_Direct_16s_I(Ipp16s* pSrcDstVal, const Ipp16s* pTaps, int
    order, Ipp32s* pDlyLine);
```

Parameters

<i>src</i>	Input sample.
<i>pDstVal</i>	Pointer to the output filtered sample.
<i>pSrcDstVal</i>	Pointer to the input and output sample for in-place operation.
<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $2 \cdot (\text{order} + 1)$.
<i>order</i>	Order of the arbitrary IIR filter.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>order</i> .

Description

The function `ippsIIROne_Direct` is declared in the `ipps.h` file. This function filters a single sample *src* (*pSrcDstVal* for in-place flavor) through an arbitrary order IIR filter and stores the result in *pDstVal* (*pSrcDstVal*). The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The *order*-length array *pDlyLine* specifies the delay line values. The $2 \cdot (\text{order} + 1)$ -length array *pTaps* specifies the taps arranged in the array as follows:

$$B_0, B_1, \dots, B_{\text{order}}, A_0, A_1, \dots, A_{\text{order}}$$

The value $A_0 \geq 0$ holds the scale factor (and not a divisor) for all the other taps.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsIIROrderErr</code>	Indicates an error when <i>order</i> is negative .
<code>ippStsScaleRangeErr</code>	Indicates an error when A_0 is less than 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect

IIROne_BiQuadDirect

Directly filters a single sample through the biquad IIR filter.

Syntax

```
IppStatus ippSIIROne_BiQuadDirect_16s(Ipp16s src, Ipp16s* pDstVal, const
    Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
IppStatus ippSIIROne_BiQuadDirect_16s_I(Ipp16s* pSrcDstVal, const Ipp16s*
    pTaps, int numBq, Ipp32s* pDlyLine);
```

Parameters

<i>src</i>	Input sample.
<i>pDstVal</i>	Pointer to the output filtered sample.
<i>pSrcDstVal</i>	Pointer to the input and output sample for in-place operation.
<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $6 * numBq$.
<i>numBq</i>	Number of cascades of biquads.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * numBq$.

Description

The function `ippSIIROne_BiQuadDirect` is declared in the `ipps.h` file. This function filters a single sample *src* (*pSrcDstVal* for in-place flavors) through a biquad IIR filter and stores the result in *pDstVal* (*pSrcDstVal*). The number of cascades of biquads is defined by the *numBq* value. The $2 * numBq$ -length array *pDlyLine* specifies the delay line values. The $6 * numBq$ -length array *pTaps* specifies the taps arranged in the array as follows:

$$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \dots A_{numBq-1,2}$$

Values $A_{n,0} \geq 0$, $B_{n,0} \geq 0$ hold the scale factors (and not divisors) for all the other taps of each section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>numBq</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when $A_{n,0}$ is less than 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect

IIR_Direct

Directly filters a block of samples through the arbitrary IIR filter.

Syntax

```
IppStatus ippSIIR_Direct_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, const
    Ipp16s* pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippSIIR_Direct_16s_I(Ipp16s* pSrcDst, int len, const Ipp16s* pTaps,
    int order, Ipp32s* pDlyLine);
```

Parameters

<i>pSrc</i>	Pointer to the input array of samples.
<i>pDst</i>	Pointer to the output array of filtered samples.
<i>pSrcDst</i>	Pointer to the input and output array for the in-place operation.
<i>len</i>	Number of samples to be filtered.
<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $2*(order+1)$.
<i>order</i>	Order of the arbitrary IIR filter.

pDlyLine Pointer to the array containing the delay line values. The number of elements in the array is *order*.

Description

The function `ippsIIR_Direct` is declared in the `ipps.h` file. This function filters *len* samples in the input array *pSrc* or *pSrcDst* through an arbitrary IIR filter and stores the results in *pDst* or *pSrcDst*, respectively. The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The *order* -length array *pDlyLine* specifies the delay line values. The $2 \cdot (\text{order} + 1)$ -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \dots, B_{\text{order}}, A_0, A_1, \dots, A_{\text{order}}$

The value $A_0 \geq 0$ holds the scale factor (and not a divisor) for all the other taps.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>order</i> is negative.
<code>ippStsScaleRangeErr</code>	Indicates an error when A_0 is less than 0.

IIR_BiQuadDirect

Directly filters a block of samples through the biquad IIR filter.

Syntax

```
IppStatus ippsIIR_BiQuadDirect_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    const Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
IppStatus ippsIIR_BiQuadDirect_16s_I(Ipp16s* pSrcDst, int len, const Ipp16s*
    pTaps, int numBq, Ipp32s* pDlyLine);
```

Parameters

<i>pSrc</i>	Pointer to the input array of samples.
<i>pDst</i>	Pointer to the output array of filtered samples.
<i>pSrcDst</i>	Pointer to the input and output array for the in-place operation.
<i>len</i>	Number of samples to be filtered.
<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $6 * numBq$ for BQ filters.
<i>numBq</i>	Number of cascades of biquads.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * numBq$ for BQ filters.

Description

The function `ippsIIR_BiQuadDirect` is declared in the `ipps.h` file. This function filters *len* samples in the input array *pSrc* or *pSrcDst* through a biquad IIR filter and stores the result in *pDst* or *pSrcDst*, respectively. The number of cascades of biquads is defined by the *numBq* value. The $2 * numBq$ -length array *pDlyLine* specifies the delay line values. The $6 * numBq$ -length array *pTaps* specifies the taps arranged in the array as follows:

$$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \dots A_{numBq-1,2}$$

Values $A_{n,0} \geq 0$, $B_{n,0} \geq 0$ hold the scale factors (and not divisors) for all the other taps of each section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>numBq</i> is less than 1.
<code>ippStsScaleRangeErr</code>	Indicates an error when $A_{n,0}$ is less than 0.

IIRSparseInit

Initializes a sparse IIR filter structure.

Syntax

```
IppStatus ippsIIRSparseInit_32f(IppsIIRSparseState_32f** ppState, const
    Ipp32f* pNZTaps, const Ipp32s* pNZTapPos, int nzTapsLen1, int
    nzTapsLen2, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
```

Parameters

<i>pNZTaps</i>	Pointer to the array containing the non-zero tap values.
<i>pNZTapPos</i>	Pointer to the array containing positions of the non-zero tap values. The number of elements in the array is <i>nzTapsLen</i> .
<i>nzTapsLen1, nzTapsLen2</i>	Number of elements in the array containing the non-zero tap values.
<i>pDlyLine</i>	Pointer to the array containing the delay line values.
<i>ppState</i>	Double pointer to the sparse IIR state structure.
<i>pBuffer</i>	Pointer to the external buffer for the sparse IIR state structure.

Description

The function `ippsIIRSparseInit` is declared in the `ipps.h` file. This function initializes a sparse IIR filter state structure *pState* in the external buffer *pBuffer*. The size of this buffer should be computed previously by calling the function [ippsIIRSparseGetStateSize](#).

The $(nzTapsLen1 + nzTapsLen2)$ -length array *pNZTaps* specifies the non-zero taps arranged in the array as follows:

$$B_0, B_1, \dots, B_{nzTapsLen1-1}, A_0, A_1, \dots, A_{nzTapsLen2-1}.$$

The $(nzTapsLen1 + nzTapsLen2)$ -length array *pNZTapPos* specifies the non-zero tap positions arranged in the array as follows:

$$BP_0, BP_1, \dots, BP_{nzTapsLen1-1}, AP_0, AP_1, \dots, AP_{nzTapsLen2-1}, \quad AP_0 \neq 0$$

The initialization function copies the values of filter coefficients from the array *pNZTaps* containing non-zero taps and their positions from the array *pNZTapPos* into the state structure *pState*.

The array *pDlyLine* contains the delay line values. The number of elements in this array is *pNZTapPos[nzTapsLen1-1] + pNZTapPos[nzTapsLen1 + nzTapsLen2 - 1]*. If the pointer to the array *pDlyLine* is not NULL, the array contents is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.



CAUTION. The values of *nzTapsLen1*, *nzTapsLen2*, *pNZTapPos[nzTapsLen - 1]*, and *pNZTapPos[nzTapsLen1 + nzTapsLen2 - 1]* should be equal to those specified in the [ippsIIRSparseGetStateSize](#) function.

Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error if one of the pointers <i>ppState</i> , <i>pNZTaps</i> , <i>pNZTapPos</i> , or <i>pBuffer</i> is NULL.
<i>ippStsIIROrderErr</i>	Indicates an error if <i>nzTapsLen1</i> is less than or equal to 0, or <i>nzTapsLen2</i> is less than 0.
<i>ippStsSparseErr</i>	Indicates an error if positions of the non-zero taps are not in ascending order, or are negative or repetitive; or <i>pNZTapPos[nzTapsLen1]</i> is equal to 0.

IIRSparseGetStateSize

Computes the size of the external buffer for the sparse IIR filter structure.

Syntax

```
IppStatus ippsIIRSparseGetStateSize_32f(int nzTapsLen1, int nzTapsLen2,
    int order1, int order2, int* pStateSize);
```

Parameters

<i>nzTapsLen1, nzTapsLen2</i>	Number of elements in the array containing the non-zero tap values.
<i>order1, order2</i>	Order of the sparse IIR filter.
<i>pStateSize</i>	Pointer to the computed value of the external buffer.

Description

The function `ippsIIRSparseGetStateSize` is declared in the `ipps.h` file. This function computes the size in bytes of the external buffer for a sparse IIR filter state that is required for the function [ippsIIRSparseInit](#). The computations are based on the specified number of non-zero filter coefficients *nzTapsLen1*, *nzTapsLen2* and filter orders *order1*, *order2*.

order1 = `pNZTapPos[nzTapsLen1-1]`,

order1 = `pNZTapPos[nzTapsLen1 + nzTapsLen2 - 1]` (see description of the function [ippsIIRSparseInit](#)). The result value is stored in the *pStateSize*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pStateSize</i> pointer is NULL.
<code>ippStsIIROrderErr</code>	Indicates an error if <i>nzTapsLen1</i> is less than or equal to 0, or <i>nzTapsLen2</i> is less than 0.
<code>ippStsSparseErr</code>	Indicates an error if <i>order1</i> or <i>order2</i> is less than 0.

IIRSparse

Filters a block of samples through a sparse IIR filter.

Syntax

```
IppStatus ippsIIRSparse_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    IppsIIRSparseState_32f* pState);
```

Parameters

<i>pState</i>	Pointer to the sparse IIR filter state structure.
<i>pSrc</i>	Pointer to the input array.

<i>pDst</i>	Pointer to the output array.
<i>len</i>	Number of elements that will be filtered.

Description

The function `ippsIIRSparse` is declared in the `ipps.h` file. This function applies the sparse IIR filter to the *len* elements of the input array *pSrc*, and stores the results in *pDst*. The filter parameters - the number of non-zero taps *nzTapsLen1*, *nzTapsLen2*, their values *pNZTaps* and their positions *pNZTapPos*, and the delay line values *pDlyLine* - are specified in the sparse IIR filter structure *pState* that should be previously initialized the function [ippsIIRSparseInit](#).

In the following definition of the sparse IIR filter, the sample to be filtered is denoted $x(n)$, the non-zero taps are denoted B_i and A_i , their positions are denoted BP_i and AP_i .

The non-zero taps are arranged in the array as follows:

$$B_0, B_1, \dots, B_{nzTapsLen1-1}, A_0, A_1, \dots, A_{nzTapsLen2-1}.$$

The non-zero tap positions are arranged in the array as follows:

$$BP_0, BP_1, \dots, BP_{nzTapsLen1-1}, AP_0, AP_1, \dots, AP_{nzTapsLen2-1}, \quad AP_0 \neq 0$$

The return value is $y(n)$ is defined by the formula for a sparse IIR filter:

$$y(n) = \sum_{k=0}^{nzTapsLen1-1} B_k \cdot x(n-BP_k) + \sum_{k=1}^{nzTapsLen2-1} A_k \cdot y(n-AP_k) \quad 0 \leq n < len$$

After the function has performed calculations, it updates the delay line values stored in the filter state structure.

[Example 6-9](#) shows how to use the sparse IIR filter functions.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippsStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.

Example 6-9 Using the Sparse IIR Filter Functions

```

int buflen;
Ipp8u *buf;
int nzTapsLen1 = 5;//number of non-zero taps in the FIR part of the formula
int nzTapsLen2 = 3;//number of non-zero taps in the IIR part of the formula
Ipp32f nzTaps [] = {0.5, 0.4, 0.3, 0.2, 0.1, 0.8, 0.7, 0.6};
                                //non-zero taps values (FIR+IIR)
Ipp32s nzTapsPos[] = {0, 10, 20, 30, 40, 1, 5, 15};
                                //non-zero tap positions (FIR+IIR)

IppsIIRSparseState_32f* iirState;
Ipp32f *src, *dst;

/* ..... */

ippsIIRSparseGetStateSize_32f(nzTapsLen1, nzTapsLen2, nzTapsPos [nzTapsLen1 - 1],
                               nzTapsPos [nzTapsLen1 + nzTapsLen2 - 1], &buflen);
buf = ippsMalloc_8u(buflen);
ippsIIRSparseInit_32f(&iirState, nzTaps, nzTapsPos, nzTapsLen1, nzTapsLen2,
                     NULL, buf);

/* . . . . initializing src somehow . . . . */

ippsIIRSparse_32f(src, dst, len, iirState);

/* dst[i] = src[i] * 0.5 + src[i-10] * 0.4 + src[i-20] * 0.3 + src[i-30] * 0.2
+ src[i-40] * 0.1 + dst[i-1] * 0.8 + dst[i-5] * 0.7 + dst[i-15] * 0.6 */

/* ..... */

ippsFree(buf);

```

Median Filter Functions

Median filters are nonlinear rank-order filters based on replacing each element of the input array with the median value, taken over the fixed neighborhood (mask) of the processed element. These filters are extensively used in image and signal processing applications. Median filtering removes impulsive noise, while keeping the signal blurring to the minimum. Typically mask size (or window width) is set to odd value which ensures simple function implementation and low output signal bias. In the Intel IPP median function implementation, the mask is always centered at the input element for which the median value is computed. You can use an even mask size in function calls as well, but internally it will be changed to odd by subtracting 1. Another specific feature of the median function implementation in the Intel IPP is that elements outside the input array, which are needed to determine the median value for “border” elements, are set to be equal to the corresponding edge element of the input array, i.e. are padded by cloning the edge element.

FilterMedian

Computes median values for each input array element.

Syntax

```
IppStatus ippsFilterMedian_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int
    maskSize);
IppStatus ippsFilterMedian_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, int
    maskSize);
IppStatus ippsFilterMedian_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, int
    maskSize);
IppStatus ippsFilterMedian_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, int
    maskSize);
IppStatus ippsFilterMedian_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, int
    maskSize);

IppStatus ippsFilterMedian_8u_I(Ipp8u* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_16s_I(Ipp16s* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_32s_I(Ipp32s* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_32f_I(Ipp32f* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_64f_I(Ipp64f* pSrcDst, int len, int maskSize);
```


Parameters

<i>pSrcDst</i>	Pointer to the input and output array (for the in-place operation).
<i>pSrc</i>	Pointer to the input array to be filtered by the function <code>ippsFilterMedian</code> .
<i>pDst</i>	Pointer to the output array filtered by the function <code>ippsFilterMedian</code> .
<i>len</i>	Number of elements in the array.
<i>maskSize</i>	Median mask size, must be a positive integer. If an even value is specified, the function subtracts 1 and uses the odd value of the filter mask for median filtering.

Description

The function `ippsFilterMedian` is declared in the `ipps.h` file. This function computes median values for each element of the input array *pSrc* or *pSrcDst*, and stores the result in *pDst* or *pSrcDst*, respectively.



NOTE. The value of a non-existent point is equal to the last point value, for example: $x[-1]=x[0]$ or $x[len]=x[len-1]$.

[Example 6-10](#) illustrates using `ippsFilterMedian_16s_I` for single-rate filtering.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsEvenMedianMaskSize</code>	Indicates a warning when the median mask length is even.

Example 6-10 Single-Rate Filtering with the `ippsFilterMedian` Function

```
void median(void) {  
    Ipp16s x[8] = {1,2,127,4,5,0,7,8};  
    IppStatus status = ippsFilterMedian_16s_I(x, 8, 3);  
    printf_16s("median =", x,8, status);  
}
```

Output:

```
median =  1 2 4 5 4 5 7 8
```

Matlab* Analog:

```
>> x = [1 2 127 4 5 0 7 8]; medfilt1(x)
```

Transform Functions

7

This chapter describes Intel[®] IPP functions that perform Fourier and discrete cosine transforms (DCT), as well as Hilbert and wavelet transforms of signals.

The full list of functions in this group is given in [Table 7-1](#).

Table 7-1 Intel IPP Transform Functions

Function Base Name	Operation
Support Functions	
ConjPack	Converts the data in <code>Pack</code> format to complex data format.
ConjPerm	Converts the data in <code>Perm</code> format to complex data format.
ConjCcs	Converts the data in <code>CCS</code> format to complex data format.
MulPack	Multiplies the elements of two vectors stored in <code>Pack</code> format.
MulPerm	Multiplies the elements of two vectors stored in <code>Perm</code> format.
MulPackConj	Multiplies elements of a vector by the elements of a complex conjugate vector stored in <code>Pack</code> format.
Fast Fourier Transform Functions	
FFTInitAlloc_R, FFTInitAlloc_C	Allocates memory and initializes a fast Fourier transform specification structure for real and complex signals.
FFTFree_R, FFTFree_C	Closes a fast Fourier transform structure for real and complex signals.
FFTInit_R, FFTInit_C	Initializes a fast Fourier transform specification structure for real and complex signals.
FFTGetSize_R, FFTGetSize_C	Computes sizes of the FFT specification structure and required working buffers.
FFTGetBufSize_R, FFTGetBufSize_C	Computes the size of the FFT work buffer.

Table 7-1 Intel IPP Transform Functions (continued)

Function Base Name	Operation
<u>FFTFwd_CToC</u>	Computes the forward fast Fourier transform (FFT) of a complex signal.
<u>FFTInv_CToC</u>	Computes the inverse fast Fourier transform (FFT) of a complex signal.
<u>FFTFwd_RToPack,</u> <u>FFTFwd_RToPerm,</u> <u>FFTFwd_RToCCS</u>	Computes the forward fast Fourier transform (FFT) of a real signal.
<u>FFTInv_PackToR,</u> <u>FFTInv_PermToR,</u> <u>FFTInv_CCSToR</u>	Computes the inverse fast Fourier transform (FFT) of a real signal.
Discrete Fourier Transform Functions	
<u>DFTInitAlloc_R,</u> <u>DFTInitAlloc_C</u>	Initializes the discrete Fourier transform structure for real and complex signals.
<u>DFTFree_R, DFTFree_C</u>	Closes the discrete Fourier transform structure for real and complex signals.
<u>DFTGetBufSize_R,</u> <u>DFTGetBufSize_C</u>	Computes the size of the discrete Fourier transform work buffer.
<u>DFTFwd_CToC</u>	Computes the forward discrete Fourier transform of a complex signal.
<u>DFTInv_CToC</u>	Computes the inverse discrete Fourier transform of a complex signal.
<u>DFTFwd_RToPack,</u> <u>DFTFwd_RToPerm,</u> <u>DFTFwd_RToCCS</u>	Computes the forward discrete Fourier transform of a real signal.
<u>DFTInv_PackToR,</u> <u>DFTInv_PermToR,</u> <u>DFTInv_CCSToR</u>	Computes the inverse discrete Fourier transform of a real signal.
<u>DFTOutOrdInitAlloc_C</u>	Initializes the out-of-order discrete Fourier transform structure.
<u>DFTOutOrdFree_C</u>	Closes the out-of-order discrete Fourier transform structure.
<u>DFTOutOrdGetBufSize_C</u>	Computes the size of the work buffer for the out-of-order discrete Fourier transform.
<u>DFTOutOrdFwd_CToC</u>	Computes the forward out-of-order discrete Fourier transform.
<u>DFTOutOrdInv_CToC</u>	Computes the inverse out-of-order discrete Fourier transform.
<u>Goertz</u>	Computes the discrete Fourier transform for a given frequency for a single complex signal.

Table 7-1 Intel IPP Transform Functions (continued)

Function Base Name	Operation
<u>GoertzTwo</u>	Computes two discrete Fourier transforms for a given frequency for a single complex signal.
DCT Functions	
<u>DCTFwdInitAlloc,</u> <u>DCTInvInitAlloc</u>	Initializes the discrete cosine transform structure.
<u>DCTFwdFree,</u> <u>DCTInvFree</u>	Closes a discrete cosine transform structure.
<u>DCTFwdGetBufSize,</u> <u>DCTInvGetBufSize</u>	Computes the size of the DCT work buffer.
<u>DCTFwd,</u> <u>DCTInv</u>	Computes the forward or inverse discrete cosine transform (DCT) of a signal.
Hilbert Transform Functions	
<u>HilbertInitAlloc</u>	Initializes the Hilbert transform structure.
<u>HilbertFree</u>	Closes a Hilbert transform structure.
<u>Hilbert</u>	Computes an analytic signal using the Hilbert transform.
Wavelet Transform Functions	
<u>WTHaarFwd,</u> <u>WTHaarInv</u>	Performs forward or inverse single-level discrete wavelet Haar transforms.
<u>WTFwdInitAlloc,</u> <u>WTInvInitAlloc</u>	Initializes the wavelet transform structure.
<u>WTFwdFree,</u> <u>WTInvFree</u>	Closes a wavelet transform structure.
<u>WTFwd</u>	Computes the forward wavelet transform.
<u>WTFwdSetDlyLine,</u> <u>WTFwdGetDlyLine</u>	Sets and gets the delay lines of the forward wavelet transform.
<u>WTInv</u>	Computes the inverse wavelet transform.
<u>WTInvSetDlyLine,</u> <u>WTInvGetDlyLine</u>	Sets and gets the delay lines of the inverse wavelet transform.

Fourier Transform Functions

This section describes the Fourier and the discrete cosine transform functions in the Intel IPP. The Intel IPP contains functions which perform the discrete Fourier transform (DFT), the fast Fourier transform (FFT), and the discrete cosine transform (DCT) of signal samples. It also includes variations of the basic functions to support different application requirements.

Special Arguments

This section describes the *flag* and *hint* arguments used by the Fourier transform functions.

Flag and Hint Arguments

The Fourier transform functions require you to specify the *flag* and *hint* arguments.

The *flag* argument specifies the result normalization method. [Table 7-2](#) lists the values you can enter for the *flag* argument. Specify one and only one of the represented values in the *flag* argument. The *A* and *B* factors are multipliers used in the DFT computation.

Table 7-2 Flag Arguments for Fourier Transform Functions

Value	A	B	Description
IPP_FFT_DIV_FWD_BY_N	1/N	1	Forward transform is done with the 1/N normalization.
IPP_FFT_DIV_INV_BY_N	1	1/N	Inverse transform is done with the 1/N normalization.
IPP_FFT_DIV_BY_SQRTN	1/N ^{1/2}	1/N ^{1/2}	Forward and inverse transform is done with the 1/N ^{1/2} normalization.
IPP_FFT_NODIV_BY_ANY	1	1	Forward or inverse transform is done without the 1/N or 1/N ^{1/2} normalization.

The *hint* argument suggests using special code, faster but less accurate calculation, or more accurate but slower calculation. [Table 7-3](#) lists values you can enter for the *hint* argument.

Note that the *hint* argument is also used for tone generation, see [Tone-Generating Functions](#) in Chapter 4, and for logarithmic addition, see [Model Evaluation](#) in Chapter 8.

Table 7-3 Hint Arguments for Fourier Transform Functions

Value	Description
<code>ippAlgHintNone</code>	No suggestions. The function selects an algorithm
<code>ippAlgHintFast</code>	Fast algorithm is suggested for computation.
<code>ippAlgHintAccurate</code>	Accurate algorithm is suggested for computation.

Packed Formats

This section describes the main packed formats `Perm`, `Pack`, and `CCS` used by the Fourier transform functions.

Pack Format

The `Pack` format is a convenient, compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real Fourier transform algorithms (“natural” in the sense that bit-reversed order is natural for radix-2 complex Fourier transforms). In `Pack` format, the output samples of the Fourier transform are arranged as shown in [Table 7-4](#) and [Table 7-5](#). The output signal can be unpacked to a complex signal using the function [`ippsConjPack`](#).

Perm Format

The `Perm` format stores the values in the order in which the Fourier transform algorithms use them. This is the most natural way of storing values for the Fourier transform algorithms. The `Perm` format is an arbitrary permutation of the `Pack` format. An important characteristic of the `Perm` format is that the real and imaginary parts of a given sample need not be adjacent.

In `Perm` format, the output samples of the Fourier transform are arranged as shown in [Table 7-4](#) and [Table 7-5](#). The output signal can be unpacked to a complex signal using the function [`ippsConjPerm`](#).



NOTE. For input signal of odd length the `perm` and `pack` format are identical.

CCS Format

The CCS format stores the values of the first half of the output complex signal resulted from the forward Fourier transform.

In CCS format, the output samples of the Fourier transform are arranged as shown in [Table 7-4](#) and [Table 7-5](#). Note that the signal stored in CCS format is one complex element longer. The output signal can be unpacked to a complex signal using the function [ippsConjCCs](#).

Table 7-4 Arrangement of Forward Fourier Transform Results in Packed Formats - Even Length

Index	0	1	2	3	...	N-2	N-1	N	N+1
Pack	R_0	R_1	I_1	R_2	...	$I_{(N-1)/2}$	$R_{N/2}$		
Perm	R_0	$R_{N/2}$	R_1	I_1	...	$R_{N/2-1}$	$I_{N/2-1}$		
CCS	R_0	0	R_1	I_1	...	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

Table 7-5 Forward Fourier transform Result Representation in Packed Formats - Odd Length

Index	0	1	2	3	...	N-2	N-1	N
Pack	R_0	R_1	I_1	R_2	...	$R_{(N-1)/2}$	$I_{(N-1)/2}$	
Perm	R_0	R_1	I_1	R_2	...	$R_{(N-1)/2}$	$I_{(N-1)/2}$	
CCS	R_0	0	R_1	I_1	...	$I_{(N-1)/2-1}$	$R_{(N-1)/2}$	$I_{(N-1)/2}$

Format Conversion Functions

The following functions `ConjPack`, `ConjPerm`, and `ConjCCs` convert data from the packed formats to a usual complex data format using the FFT symmetry property for transforming real data. The output data is complex, the output array length is defined by the number of complex elements in the output vector. Note that the output array size is two times as big as the input array size. The data stored in CCS format require a bigger array than the other formats. Even and odd length arrays have some specific features discussed for each function separately.

ConjPack

Converts the data in Pack format to complex data format.

Syntax

```
IppStatus ippsConjPack_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPack_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPack_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPack_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPack_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPack_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>lenDst</i>	Number of elements in the vector.

Description

The function `ippsConjPack` is declared in the `ipps.h` file. This function converts the data in [Pack](#) format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjPack` converts the data in Pack format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

[Table 7-6](#) shows the examples of unpack from the Pack format. The Data column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the Packed column. The output result is the complex data vector in the Extended column. The number of vector elements is in the Length column. [Example 7-1](#) shows how to use the function `ippsConjPack`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> , <i>pDst</i> , or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>lenDst</i> is less than or equal to 0.

Table 7-6 Examples of Unpack from the Pack Format

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -2, 7, -7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-1 Using the `ippsConjPack` Function

```
void ConjPack(void) {
    Ipp16s x[8] = {1,2,3,5,6,7,8,9};
    Ipp16sc zero={0,0}, y[6];
    IppStatus st;
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjPack_16sc( x, y, 6 );
    printf_16sc("pack 6", y, 6, st );
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjPack_16sc( x, y, 5 );
    printf_16sc("pack 5", y, 5, st );
}

Output:
Pack 6: {1,0} {2,3} {5,6} {7,0} {5,-6} {2,-3}
Pack 5: {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjPerm

Converts the data in Perm format to complex data format.

Syntax

```
IppStatus ippsConjPerm_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPerm_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPerm_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPerm_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>lenDst</i>	Number of elements in the vector.

Description

The function `ippsConjPerm` is declared in the `ipps.h` file. This function converts the data in [Perm](#) format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjPerm` converts the data in Perm format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

[Table 7-7](#) shows the examples of unpack from the Perm format. The Data column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the Packed column. The output result is the complex data vector in the Extended column. The number of vector elements is in the Length column. [Example 7-2](#) shows how to use the function `ippsConjPerm`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

ippStsNullPtrErr

Indicates an error when the *pSrcDst*, *pDst*, or *pSrc* pointer is NULL.

ippStsSizeErr

Indicates an error when *lenDst* is less than or equal to 0.

Table 7-7 **Examples of Packed Data Obtained by FFT**

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -7, -2, 7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-2 **Using the `ippsConjPerm` Function**

```
void ConjPerm(void) {
    Ipp16s x[8] = {1,2,3,5,6,7,8,9};
    Ipp16sc zero={0,0}, y[6];
    IppStatus st;
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjPerm_16sc( x, y, 6 );
    printf_16sc("Perm 6:", y, 6, st );
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjPerm_16sc( x, y, 5 );
    printf_16sc("Perm 5:", y, 5, st );
}

Output:
Perm 6:  {1,0} {3,5} {6,7} {2,0} {6,-7} {3,-5}
Perm 5:  {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjCcs

Converts the data in CCS format to complex data format.

Syntax

```
IppStatus ippsConjCcs_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjCcs_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjCcs_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjCcs_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>lenDst</i>	Number of elements in the vector.

Description

The function `ippsConjCcs` is declared in the `ipps.h` file. This function converts the data in [CCS](#) format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjCcs` converts the data in [CCS](#) format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

[Table 7-8](#) shows the examples of unpack from the CCS format. The Data column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the Packed column. The output result is the complex data vector in the Extended column. The number of vector elements is in the Length column. The data stored in CCS format are two real elements longer. [Example 7-3](#) shows how to use the function `ippsConjCcs`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when the `pSrcDst`, `pDst`, or `pSrc` pointer is NULL.

`ippStsSizeErr` Indicates an error when `lenDst` is less than or equal to 0.

Table 7-8 Examples of Unpack from the CCS Format

Data	Packed	Extended	Length
FFT([1])	1, 0	{1, 0}	1
FFT([1 2])	3, 0, -1, 0	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, 0, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, 0, -2, 7, -7, 0	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-3 Using the `ippsConjCcs` Function

```
void ConjCcs(void) {
    Ipp16s x[8] = {1,2,3,5,6,7,8,9};
    Ipp16sc zero={0,0}, y[6];
    IppStatus st;
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjCcs_16sc( x, y, 6 );
    printf_16sc("CCS 6:", y, 6, st );
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjCcs_16sc( x, y, 5 );
    printf_16sc("CCS 5:", y, 5, st );
}

Output:
CCS 6:  {1,2} {3,5} {6,7} {8,9} {6,-7} {3,-5}
CCS 5:  {1,2} {3,5} {6,7} {6,-7} {3,-5}
```

Functions for Packed Data Multiplication

The functions described in this section perform the element-wise complex multiplication of vectors stored in [Pack](#) or [Perm](#) formats. These functions are used with the function `ippsFFTFwd` and `ippsFFTInv` to perform fast convolution on real signals.

The standard vector multiplication function `ippsMul` can not be used to multiply `Pack` or `Perm` format vectors because:

- Two real samples are stored in `Pack` format.
- The `Perm` format might not pair the real parts of a signal with their corresponding imaginary parts.



NOTE. The vectors stored in [CCS](#) format can be multiplied using the standard function for complex data multiplication.

MulPack

Multiply the elements of two vectors stored in `Pack` format.

Syntax

Case 1: Not-in-place operation.

```
IppStatus ippsMulPack_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int length);
IppStatus ippsMulPack_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int length);
IppStatus ippsMulPack_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int length, int scaleFactor);
```

Case 2: In-place operation.

```
IppStatus ippsMulPack_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int length);
IppStatus ippsMulPack_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int length);
IppStatus ippsMulPack_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int length,
    int scaleFactor);
```

Parameters

`pSrc1`, `pSrc2` Pointers to the vectors whose elements are to be multiplied together.

<i>pDst</i>	Pointer to the destination vector which stores the result of the multiplication $pSrc1[n] * pSrc2[n]$.
<i>pSrc</i>	Pointer to the vector whose elements are to be multiplied by the elements of <i>pSrcDst</i> in-place.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>length</i>	Number of elements in the vector.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMulPack` is declared in the `ipps.h` file. This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2*, and store the result in *pDst*.

The in-place flavors `ippsMulPack` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst*, and store the result in *pSrcDst*.

The functions multiply the packed data according to their packed format. The data in [Pack](#) packed format include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

For the functions with the `Sfs` suffixes scaling is performed in accordance with the *scaleFactor* value. When the output value exceeds the data range, the result may become saturated.

[Example 7-4](#) shows how to use the function `ippsMulPack_32f_I`.

Return Values

<code>ippStsNoErr</code>	Indicates no error
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> , <i>pDst</i> , <i>pSrc1</i> , <i>pSrc2</i> , or <i>pSrc</i> pointer is NULL
<code>ippStsSizeErr</code>	Indicates an error when <i>length</i> is less than or equal to 0

Example 7-4 Using the `ippsMulPack` Function

```
void mulpack( void ) {
    Ipp32f x[8], X[8], h[8]={1.0f/3,1.0f/3,1.0f/3,0,0,0,0,0}, H[8];
    IppStatus st;
    IppsFFTSpec_R_32f* spec;
    st = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
        ippsAlgHintNone);
    ippsSet_32f( 3, x, 8 );
    x[3] = 5;
    st = ippsFFTFwd_RToPack_32f( x, X, spec, NULL );
    st = ippsFFTFwd_RToPack_32f( h, H, spec, NULL );
    ippsMulPack_32f_I( H, X, 8 );
    st = ippsFFTInv_PackToR_32f( X, x, spec, NULL );
    printf_32f("filtered =", x, 8, st );
    ippsFFTFree_R_32f( spec );
}
```

Output:

```
filtered =  3.0 3.0 3.0 3.666667 3.666667 3.666667 3.0 3.0
```

Matlab* analog:

```
>> x=3*ones(1,8); x(4)=5;h=zeros(1,8); h(1:3)=1/3;
real(ifft(fft(x).*fft(h)))
```

MulPerm

Multiply the elements of two vectors stored in Perm format.

Syntax

Case 1: Not-in-place operation.

```
IppStatus ippsMulPerm_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int length);
```

```
IppStatus ippsMulPerm_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int length);
IppStatus ippsMulPerm_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int length, int scaleFactor);
```

Case 2: In-place operation.

```
IppStatus ippsMulPerm_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int length);
IppStatus ippsMulPerm_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int length);
IppStatus ippsMulPerm_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int length,
    int scaleFactor);
```

Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the vectors whose elements are to be multiplied together.
<i>pDst</i>	Pointer to the destination vector which stores the result of the multiplication $pSrc1[n] * pSrc2[n]$.
<i>pSrc</i>	Pointer to the vector whose elements are to be multiplied by the elements of <i>pSrcDst</i> in-place.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>length</i>	Number of elements in the vector.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMulPerm` is declared in the `ipps.h` file. This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2*, and store the result in *pDst*.

The in-place flavors of `ippsMulPerm` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst*, and store the result in *pSrcDst*.

The function multiplies the packed data according to their packed format. The data in [Perm](#) packed formats include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

For the functions with the `Sfs` suffixes scaling is performed in accordance with the *scaleFactor* value. When the output value exceeds the data range, the result may become saturated.

Return Values

<code>ippStsNoErr</code>	Indicates no error
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> , <code>pDst</code> , <code>pSrc1</code> , <code>pSrc2</code> , or <code>pSrc</code> pointer is <code>NULL</code>
<code>ippStsSizeErr</code>	Indicates an error when <code>length</code> is less than or equal to 0

MulPackConj

Multiplies elements of a vector by the elements of a complex conjugate vector stored in `Pack` format.

Syntax

```
IppStatus ippsMulPackConj_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int length);  
IppStatus ippsMulPackConj_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int length);
```

Parameters

<code>pSrc</code>	Pointer to the first source vector.
<code>pSrcDst</code>	Pointer to the second source and destination vector.
<code>length</code>	Number of elements in the vector.

Description

The function `ippsMulPackConj` is declared in the `ipps.h` file. This function multiplies the elements of a source vector `pSrc` by elements of the vector that is complex conjugate to the source vector `pSrcDst` and stores the results in `pSrcDst`. The function performs only in-place operations on data stored in [Pack](#) format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>length</code> is less than or equal to 0.

Fast Fourier Transform Functions

The functions described in this section compute the forward and inverse fast Fourier transform of real and complex signals. The FFT is similar to the discrete Fourier transform (DFT) but is significantly faster. The length of the vector transformed by the FFT must be a power of 2.

To use the FFT functions, initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms. The amount of prior calculations is thus reduced and the overall performance increased.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method.

The FFT specification structure can be initialized by the functions [ippsFFTInitAlloc_C](#) or [ippsFFTInitAlloc_R](#) that allocate memory and initialize the structure. Alternatively the structure can be initialized by the functions [ippsFFTInit_C](#) and [ippsFFTInit_R](#). In this case the user must allocate memory for the FFT specification structure whose size should be computed previously using either [ippsFFTGetSize_C](#) or [ippsFFTGetSize_R](#) functions.

The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in Perm, Pack, or CCS format.

You can speed up the FFT on Intel® processors with SIMD instructions, if assign an external buffer. The external buffer increases performance because it allows to avoid allocation and deallocation of internal buffers and to store data in cache. The size of this buffer can be previously computed by either [ippsFFTGetBufSize_C](#) or [ippsFFTGetBufSize_R](#).

FFTInitAlloc_R, FFTInitAlloc_C

Allocates memory and initializes an FFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal.

```
IppStatus ippsFFTInitAlloc_R_16s(IppsFFTSpec_R_16s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
```

```
IppStatus ippsFFTInitAlloc_R_32s(IppsFFTSpec_R_32s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_R_32f(IppsFFTSpec_R_32f** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_R_64f(IppsFFTSpec_R_64f** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_R_16s32s(IppsFFTSpec_R_16s32s** ppFFTSpec, int
    order, int flag, IppHintAlgorithm hint);
```

Case 2: Operation on complex signal.

```
IppStatus ippsFFTInitAlloc_C_16s(IppsFFTSpec_C_16s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32s(IppsFFTSpec_C_32s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32f(IppsFFTSpec_C_32f** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_64f(IppsFFTSpec_C_64f** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_C_16sc(IppsFFTSpec_C_16sc** ppFFTSpec, int order,
    int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32sc(IppsFFTSpec_C_32sc** ppFFTSpec, int order,
    int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32fc(IppsFFTSpec_C_32fc** ppFFTSpec, int order,
    int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_64fc(IppsFFTSpec_C_64fc** ppFFTSpec, int order,
    int flag, IppHintAlgorithm hint);
```

Parameters

<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in “ Flag and Hint Arguments .”
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments .”
<i>order</i>	FFT order. The input signal length is $N=2^{order}$.
<i>ppFFTSpec</i>	Pointer to the FFT specification structure to be created.

Description

The functions `ippsFFTInitAlloc_R` and `ippsFFTInitAlloc_C` are declared in the `ipps.h` file. These functions allocate memory, create, and initialize the FFT specification structure `ppFFTSpec` with the following parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The *order* argument defines the transform length. Thus the input and output signals are 2^{order} -length arrays.

ippsFFTInitAlloc_R. The function `ippsFFTInitAlloc_R` allocates memory and initializes the real FFT specification structure.

ippsFFTInitAlloc_C. The function `ippsFFTInitAlloc_C` allocates memory and initializes the complex FFT specification structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>ppFFTSpec</code> pointer is NULL.
<code>ippStsFftOrderErr</code>	Indicates an error when the <i>order</i> value is incorrect.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

FFTFree_R, FFTFree_C

Closes an FFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal.

```
IppStatus ippsFFTFree_R_16s(IppsFFTSpec_R_16s* pFFTSpec);
IppStatus ippsFFTFree_R_32s(IppsFFTSpec_R_32s* pFFTSpec);
IppStatus ippsFFTFree_R_32f(IppsFFTSpec_R_32f* pFFTSpec);
IppStatus ippsFFTFree_R_64f(IppsFFTSpec_R_64f* pFFTSpec);
IppStatus ippsFFTFree_R_16s32s(IppsFFTSpec_R_16s32s* pFFTSpec);
```

Case 2: Operation on complex signal.

```
IppStatus ippsFFTFree_C_16s(IppsFFTSpec_C_16s* pFFTSpec);
IppStatus ippsFFTFree_C_32s(IppsFFTSpec_C_32s* pFFTSpec);
IppStatus ippsFFTFree_C_32f(IppsFFTSpec_C_32f* pFFTSpec);
IppStatus ippsFFTFree_C_64f(IppsFFTSpec_C_64f* pFFTSpec);

IppStatus ippsFFTFree_C_16sc(IppsFFTSpec_C_16sc* pFFTSpec);
IppStatus ippsFFTFree_C_32sc(IppsFFTSpec_C_32sc* pFFTSpec);
IppStatus ippsFFTFree_C_32fc(IppsFFTSpec_C_32fc* pFFTSpec);
IppStatus ippsFFTFree_C_64fc(IppsFFTSpec_C_64fc* pFFTSpec);
```

Parameters

pFFTSpec Pointer to the FFT specification structure to be closed.

Description

The function `ippsFFTFree` is declared in the `ipps.h` file. This function closes the FFT specification structure *pFFTSpec* by freeing all memory associated with the specification created by [ippsFFTInitAlloc_C](#) or [ippsFFTInitAlloc_R](#). Call `ippsFFTFree` after the transform is completed.

ippsFFTFree_C. The function `ippsFFTFree_C` closes the complex FFT specification structure.

ippsFFTFree_R. The function `ippsFFTFree_R` closes the real FFT specification structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFFTSpec</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFFTSpec</i> is incorrect.

FFTInit_R, FFTInit_C

Initializes an FFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal.

```

IppStatus ippsFFTInit_R_16s(IppsFFTSpec_R_16s** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
IppStatus ippsFFTInit_R_32s(IppsFFTSpec_R_32s** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
IppStatus ippsFFTInit_R_32f(IppsFFTSpec_R_32f** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
IppStatus ippsFFTInit_R_64f(IppsFFTSpec_R_64f** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
IppStatus ippsFFTInit_R_16s32s(IppsFFTSpec_R_16s32s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);

```

Case 2: Operation on complex signal.

```

IppStatus ippsFFTInit_C_16s(IppsFFTSpec_C_16s** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
IppStatus ippsFFTInit_C_32s(IppsFFTSpec_C_32s** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
IppStatus ippsFFTInit_C_32f(IppsFFTSpec_C_32f** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
IppStatus ippsFFTInit_C_64f(IppsFFTSpec_C_64f** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);

IppStatus ippsFFTInit_C_16sc(IppsFFTSpec_C_16sc** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
IppStatus ippsFFTInit_C_32sc(IppsFFTSpec_C_32sc** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
IppStatus ippsFFTInit_C_32fc(IppsFFTSpec_C_32fc** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);

```



```
IppStatus ippsFFTInit_C_64fc(IppsFFTSpec_C_64fc** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pMemSpec, Ipp8u* pBufInit);
```

Parameters

<i>order</i>	FFT order. The input signal length is $N=2^{order}$.
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in “ Flag and Hint Arguments .”
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments .”
<i>ppFFTSpec</i>	Double pointer to the FFT specification structure to be created.
<i>pMemSpec</i>	Pointer to the area for the FFT specification structure.
<i>pBufInit</i>	Pointer to the work buffer.

Description

The functions `ippsFFTInit_R` and `ippsFFTInit_C` are declared in the `ipps.h` file. These functions initialize the FFT specification structure *ppFFTSpec* with the following parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The *order* argument defines the transform length. Thus the input and output signals are 2^{order} -length arrays.

Before calling these functions the size of the FFT specification structure should be computed by the either [ippsFFTGetSize_C](#) or [ippsFFTGetSize_R](#).

The functions require the working buffer *pBufInit* whose size should be computed by the same functions [ippsFFTGetBufSize_C](#) or [ippsFFTGetBufSize_R](#) beforehand.

ippsFFTInit_C. The function `ippsFFTInit_C` initializes the complex FFT specification structure.

ippsFFTInit_R. The function `ippsFFTInit_R` initializes the real FFT specification structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsFftOrderErr</code>	Indicates an error when the <i>order</i> value is incorrect.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.

FFTGetSize_R, FFTGetSize_C

Computes sizes of the FFT Specification structure and required working buffers.

Syntax

Case 1: Operation on real signal.

```
IppStatus ippsFFTGetSize_R_16s(int order, int flag, IppHintAlgorithm hint,
    int* pSizeSpec, int* pSizeInit, int* pSizeBuf);
IppStatus ippsFFTGetSize_R_32s(int order, int flag, IppHintAlgorithm hint, int*
    pSizeSpec, int* pSizeInit, int* pSizeBuf);
IppStatus ippsFFTGetSize_R_32f(int order, int flag, IppHintAlgorithm hint, int*
    pSizeSpec, int* pSizeInit, int* pSizeBuf);
IppStatus ippsFFTGetSize_R_64f(int order, int flag, IppHintAlgorithm hint, int*
    pSizeSpec, int* pSizeInit, int* pSizeBuf);
IppStatus ippsFFTGetSize_R_16s32s(int order, int flag, IppHintAlgorithm hint,
    int* pSizeSpec, int* pSizeInit, int* pSizeBuf);
```

Case 2: Operation on complex signal.

```
IppStatus ippsFFTGetSize_C_16s(int order, int flag, IppHintAlgorithm hint,
    int* pSizeSpec, int* pSizeInit, int* pSizeBuf);
IppStatus ippsFFTGetSize_C_32s(int order, int flag, IppHintAlgorithm hint, int*
    pSizeSpec, int* pSizeInit, int* pSizeBuf);
IppStatus ippsFFTGetSize_C_32f(int order, int flag, IppHintAlgorithm hint, int*
    pSizeSpec, int* pSizeInit, int* pSizeBuf);
IppStatus ippsFFTGetSize_C_64f(int order, int flag, IppHintAlgorithm hint, int*
    pSizeSpec, int* pSizeInit, int* pSizeBuf);

IppStatus ippsFFTGetSize_C_16sc(int order, int flag, IppHintAlgorithm hint,
    int* pSizeSpec, int* pSizeInit, int* pSizeBuf);
IppStatus ippsFFTGetSize_C_32sc(int order, int flag, IppHintAlgorithm hint,
    int* pSizeSpec, int* pSizeInit, int* pSizeBuf);
IppStatus ippsFFTGetSize_C_32fc(int order, int flag, IppHintAlgorithm hint,
    int* pSizeSpec, int* pSizeInit, int* pSizeBuf);
```

```
IppStatus ippsFFTGetSize_C_64fc(int order, int flag, IppHintAlgorithm hint,
    int* pSizeSpec, int* pSizeInit, int* pSizeBuf);
```

Parameters

<i>order</i>	FFT order. The input signal length is $N=2^{order}$.
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in “ Flag and Hint Arguments .”
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments .”
<i>pSizeSpec</i>	Pointer to the FFT specification structure size value.
<i>pSizeInit</i>	Pointer to the buffer size value for FFT initialization function.
<i>pSizeBuf</i>	Pointer to the FFT work buffer size value.

Description

The functions `ippsFFTGetSize_R` and `ippsFFTGetSize_C` are declared in the `ipps.h` file. These functions compute the size of FFT specification structure, the work buffer size for the FFT structure initialization functions [ippsFFTInit_C](#) and [ippsFFTInit_R](#) as well as the size of the FFT work buffer for the flavors of functions `ippsFFTFwd` and `ippsFFTInv`. Their values in bytes are stored in *pSizeSpec*, *pSizeInit*, and *pSizeBuf* respectively.

ippsFFTGetSize_C. The function `ippsFFTGetSize_C` computes the size of the complex FFT work buffer.

ippsFFTGetSize_R. The function `ippsFFTGetSize_R` gets the size of the real FFT work buffer.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsFftOrderErr</code>	Indicates an error if the <i>order</i> value is incorrect.
<code>ippStsFftFlagErr</code>	Indicates an error if the <i>flag</i> value is incorrect.

FFTGetBufSize_R, FFTGetBufSize_C

Computes the size of the FFT work buffer.

Syntax

Case 1: Operation on real signal.

```
IppStatus ippsFFTGetBufSize_R_16s(const IppsFFTSpec_R_16s* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_R_32s(const IppsFFTSpec_R_32s* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_R_32f(const IppsFFTSpec_R_32f* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_R_64f(const IppsFFTSpec_R_64f* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_R_16s32s(const IppsFFTSpec_R_16s32s* pFFTSpec, int*
    pSize);
```

Case 2: Operation on complex signal.

```
IppStatus ippsFFTGetBufSize_C_16s(const IppsFFTSpec_C_16s* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_C_32s(const IppsFFTSpec_C_32s* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_C_32f(const IppsFFTSpec_C_32f* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_C_64f(const IppsFFTSpec_C_64f* pFFTSpec, int* pSize);

IppStatus ippsFFTGetBufSize_C_16sc(const IppsFFTSpec_C_16sc* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_C_32sc(const IppsFFTSpec_C_32sc* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_C_32fc(const IppsFFTSpec_C_32fc* pFFTSpec, int* pSize);
IppStatus ippsFFTGetBufSize_C_64fc(const IppsFFTSpec_C_64fc* pFFTSpec, int* pSize);
```

Parameters

<i>pFFTSpec</i>	Pointer to the FFT specification structure.
<i>pSize</i>	Pointer to the FFT work buffer size value.

Description

The functions `ippsFFTGetBufSize_R` and `ippsFFTGetBufSize_C` are declared in the `ipps.h` file. These functions compute the size in bytes of the work buffer that is required for the functions that perform FFT and stores the result in `pSize`. To compute the size of the buffer the corresponding FFT specification structure `pFFTSpec` should be specified.

ippsFFTGetBufSize_C. The function `ippsFFTGetBufSize_C` computes the size of the complex FFT work buffer.

ippsFFTGetBufSize_R. The function `ippsFFTGetBufSize_R` computes the size of the real FFT work buffer.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pFFTSpec</code> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <code>pFFTSpec</code> is incorrect.

FFTFwd_CToC

Computes the forward fast Fourier transform (FFT) of a complex signal.

Syntax

Case 1: Not-in-place operation on real data type.

```
IppStatus ippsFFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u*
    pBuffer);

IppStatus ippsFFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u*
    pBuffer);

IppStatus ippsFFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsFFTSpec_C_16s* pFFTSpecx, int
    scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_32s_Sfs(const Ipp32s* pSrcRe, const Ipp32s* pSrcIm,
    Ipp32s* pDstRe, Ipp32s* pDstIm, const IppsFFTSpec_C_32s* pFFTSpec, int
    scaleFactor, Ipp8u* pBuffer);
```

Case 2: Not-in-place operation on complex data type.

```
IppStatus ippsFFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
    IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, const
    IppsFFTSpec_C_32sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: In-place operation on real data type.

```
IppStatus ippsFFTFwd_CToC_32f_I(Ipp32f* pSrcDstRe, Ipp32f* pSrcDstIm, const
    IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64f_I(Ipp64f* pSrcDstRe, Ipp64f* pSrcDstIm, const
    IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_16s_ISfs(Ipp16s* pSrcDstRe, Ipp16s* pSrcDstIm, const
    IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_32s_ISfs(Ipp32s* pSrcDstRe, Ipp32s* pSrcDstIm, const
    IppsFFTSpec_C_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 4: In-place operation on complex data type.

```
IppStatus ippsFFTFwd_CToC_32fc_I(Ipp32fc* pSrcDst, const IppsFFTSpec_C_32fc*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64fc_I(Ipp64fc* pSrcDst, const IppsFFTSpec_C_64fc*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_16sc_ISfs(Ipp16sc* pSrcDst, const IppsFFTSpec_C_16sc*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_32sc_ISfs(Ipp32sc* pSrcDst, const IppsFFTSpec_C_32sc*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

<i>pFFTSpec</i>	Pointer to the FFT specification structure.
<i>pSrc</i>	Pointer to the input array containing complex values.
<i>pDst</i>	Pointer to the output array containing complex values.

<i>pSrcRe</i>	Pointer to the input array containing real parts of the signal.
<i>pSrcIm</i>	Pointer to the input array containing imaginary parts of the signal.
<i>pDstRe</i>	Pointer to the output array containing real parts of the signal.
<i>pDstIm</i>	Pointer to the output array containing imaginary parts of the signal.
<i>pSrcDst</i>	Pointer to the input and output array containing complex values (for the in-place operation).
<i>pSrcDstRe</i>	Pointer to the input and output array containing real parts of the signal(for the in-place operation).
<i>pSrcDstIm</i>	Pointer to the input and output array containing imaginary parts of the signal(for the in-place operation).
<i>pBuffer</i>	Pointer to the FFT work buffer.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsFFTFwd_CToC` is declared in the `ipps.h` file. This function computes the forward FFT of a complex signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure should be initialized by the either [ippsFFTInitAlloc_C](#) or [ippsFFTInit_C](#).

The functions using the complex data type, e.g., with the `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*. Their in-place flavors use the complex array *pSrcDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with the `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively. Their in-place flavors uses separate real and imaginary arrays *pSrcDstRe* and *pSrcDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The functions require the working buffer *pBuffer* whose size should be computed by the functions [ippsFFTGetBufSize_C](#) or [ippsFFTGetBufSize_R](#) beforehand.

The *pBuffer* parameter provides the FFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is same for both consecutive operations.

The length of the FFT must be a power of 2.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

FFTInv_CToC

Computes the inverse fast Fourier transform (FFT) of a complex signal.

Syntax

Case 1: Not-in-place operation on real data type.

```

IppStatus ippsFFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u*
    pBuffer);

IppStatus ippsFFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u*
    pBuffer);

IppStatus ippsFFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe,
    const Ipp16s* pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm,
    const IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsFFTInv_CToC_32s_Sfs(const Ipp32s* pSrcRe, const Ipp32s* pSrcIm,
    Ipp32s* pDstRe, Ipp32s* pDstIm, const IppsFFTSpec_C_32s* pFFTSpec, int
    scaleFactor, Ipp8u* pBuffer);

```

Case 2: Not-in-place operation on complex data type.

```

IppStatus ippsFFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);

```



```
IppStatus ippsFFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
    IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, const
    IppsFFTSpec_C_32sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: In-place operation on real data type.

```
IppStatus ippsFFTInv_CToC_32f_I(Ipp32f* pSrcDstRe, Ipp32f* pSrcDstIm, const
    IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64f_I(Ipp64f* pSrcDstRe, Ipp64f* pSrcDstIm, const
    IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_16s_ISfs(Ipp16s* pSrcDstRe, Ipp16s* pSrcDstIm, const
    IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_32s_ISfs(Ipp32s* pSrcDstRe, Ipp32s* pSrcDstIm, const
    IppsFFTSpec_C_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 4: In-place operation on complex data type.

```
IppStatus ippsFFTInv_CToC_32fc_I(Ipp32fc* pSrcDst, const IppsFFTSpec_C_32fc*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64fc_I(Ipp64fc* pSrcDst, const IppsFFTSpec_C_64fc*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_16sc_ISfs(Ipp16sc* pSrcDst, const IppsFFTSpec_C_16sc*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_32sc_ISfs(Ipp32sc* pSrcDst, const IppsFFTSpec_C_32sc*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

<i>pFFTSpec</i>	Pointer to the FFT specification structure.
<i>pSrc</i>	Pointer to the input array containing complex values.
<i>pDst</i>	Pointer to the output array containing complex values.
<i>pSrcRe</i>	Pointer to the input array containing real parts of the signal.
<i>pSrcIm</i>	Pointer to the input array containing imaginary parts of the signal.
<i>pDstRe</i>	Pointer to the output array containing real parts of the signal.
<i>pDstIm</i>	Pointer to the output array containing imaginary parts of the signal.
<i>pSrcDst</i>	Pointer to the input and output array containing complex values (for the in-place operation).

<i>pSrcDstRe</i>	Pointer to the input and output array containing real parts of the signal(for the in-place operation).
<i>pSrcDstIm</i>	Pointer to the input and output array containing imaginary parts of the signal(for the in-place operation).
<i>pBuffer</i>	Pointer to the FFT work buffer.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsFFTInv_CToC` is declared in the `ipps.h` file. This function computes the inverse FFT of a complex signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure should be initialized by the either [ippsFFTInitAlloc_C](#) or [ippsFFTInit_C](#).

The functions using the complex data type, e.g., with the *32fc* suffixes, process the input complex array *pSrc* and store the result in *pDst*. Their in-place flavors use the complex array *pSrcDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with the *32f* suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively. Their in-place flavors uses separate real and imaginary arrays *pSrcDstRe* and *pSrcDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The functions require the working buffer *pBuffer* whose size should be computed by the functions [ippsFFTGetBufSize_C](#) or [ippsFFTGetBufSize_R](#) beforehand.

The *pBuffer* parameter provides the FFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is same for both consecutive operations.

The length of the FFT must be a power of 2.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFFTSpec</i> is incorrect.

`ippStsMemAllocErr`

Indicates an error when no memory is allocated.

FFTFwd_RToPack, FFTFwd_RToPerm, FFTFwd_RToCCS

Computes the forward or inverse fast Fourier transform (FFT) of a real signal.

Syntax

Case 1: Not-in-place operation, result in Pack Format.

```
IppStatus ippsFFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 2: In-place operation, result in Pack Format.

```
IppStatus ippsFFTFwd_RToPack_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: Not-in-place operation, result in Perm Format.

```
IppStatus ippsFFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 4: In-place operation, result in Perm Format.

```
IppStatus ippsFFTFwd_RToPerm_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_64f_I(Ipp64f* pSrcDst const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 5: Not-in-place operation, result in CCS Format.

```
IppStatus ippsFFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pDst,
    const IppsFFTSpec_R_16s32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 6: In-place operation, result in CCS Format.

```
IppStatus ippsFFTFwd_RToCCS_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pFFTSpec

Pointer to the FFT specification structure.

<i>pSrc</i>	Pointer to the input array.
<i>pDst</i>	Pointer to the output array containing packed complex values.
<i>pSrcDst</i>	Pointer to the input and output arrays for the in-place operation.
<i>pBuffer</i>	Pointer to the work buffer.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The functions `ippsFFTFwd_RToPerm`, `ippsFFTFwd_RToPack`, and `ippsFFTFwd_RToCCS` are declared in the `ipps.h` file. They compute the forward FFT of a real signal and store the result in [Pack](#), [Perm](#), or [CCS](#) packed formats respectively. The transform is performed in accordance with the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure should be initialized by the corresponding flavors of either [ippsFFTInitAlloc](#) or [ippsFFTInit](#). The length of the FFT must be a power of 2.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

These functions require the working buffer *pBuffer* whose size should be computed by the functions [ippsFFTGetBufSize_C](#) or [ippsFFTGetBufSize_R](#) beforehand.

The *pBuffer* argument provides the FFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

`ippsFFTFwd_RToPack`. This function computes the forward FFT and stores the result in `Pack` format.

`ippsFFTFwd_RToPerm`. This function computes the forward FFT and stores the result in `Perm` format.

`ippsFFTFwd_RToCCS`. This function computes the forward FFT and stores the result in `CCS` format.

[Table 7-4](#) and [Table 7-5](#) show how the output result is represented in each of the packed format. The code [Example 7-5](#) shows how to initialize the specification structure and call the function `ippsFFTFwd_RToCCS_32f`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

Example 7-5 Using the `ippsFFTFwd_RToCCS` Function

```

IppStatus fft( void ) {
    Ipp32f x[8], X[10];
    int n;
    IppStatus status;
    IppsFFTSpec_R_32f* spec;
    status = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
        ippAlgHintNone );
    for(n=0; n<8; ++n) x[n] = (float)cos(IPP_2PI *n *16/64);
    status = ippsFFTFwd_RToCCS_32f( x, X, spec, NULL );
    ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
    ippsFFTFree_R_32f( spec );
    printf_32f("fft magn =", x, 4, status );
    return status;
}

```

Output:

```
fft magn = 0.000000 0.000000 4.000000 0.000000
```

Matlab* Analog:

```
>> n=0:7; x=sin(2*pi*n*16/64); X=abs(fft(x)); X(1:4)
```

FFTInv_PackToR, FFTInv_PermToR, FFTInv_CCSToR

Computes the inverse fast Fourier transform (FFT) of a real signal.

Syntax

Case 1: Not-in-place operation on input data in Pack format.

```
IppStatus ippsFFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 2: In-place operation on input data in Pack format.

```
IppStatus ippsFFTInv_PackToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: Not-in-place operation on input data in Perm format.

```
IppStatus ippsFFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 4: In-place operation on input data in Perm format.

```
IppStatus ippsFFTInv_PermToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_32s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 5 Not-in-place operation on input data in CCS format.

```
IppStatus ippsFFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst,
    const IppsFFTSpec_R_16s32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 6: In-place operation on input data in CCS format.

```
IppStatus ippsFFTInv_CCSToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

<i>pFFTSpec</i>	Pointer to the FFT specification structure.
<i>pSrc</i>	Pointer to the input array.

<i>pDst</i>	Pointer to the output array containing real values.
<i>pSrcDst</i>	Pointer to the input and output for the in-place operation.
<i>pBuffer</i>	Pointer to the work buffer.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The functions `ippsFFTInv_PermToR`, `ippsFFTInv_PackToR`, and `ippsFFTInv_CCSToR` are declared in the `ipps.h` file. They compute the inverse FFT of data stored in [Pack](#), [Perm](#), or [CCS](#) packed formats respectively. The transform is performed in accordance with the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure should be initialized by the corresponding flavors of either [ippsFFTInitAlloc](#) or [ippsFFTInit](#). The length of the FFT must be a power of 2.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

These functions require the working buffer *pBuffer* whose size should be computed by the corresponding flavors of [ippsFFTGetBufSize_C](#) or [ippsFFTGetBufSize_R](#) beforehand. The *pBuffer* argument provides the FFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

ippsFFTInv_PackToR. This function computes the inverse FFT of input data in `Pack` format.

ippsFFTInv_PermToR. This function computes the inverse FFT of input data in `Perm` format.

ippsFFTInv_CCSToR. This function computes the inverse FFT of input data in `CCS` format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

Discrete Fourier Transform Functions

The functions described in this section compute the forward and inverse discrete Fourier transform of real and complex signals. The DFT is less efficient than the fast Fourier transform, however the length of the vector transformed by the DFT can be arbitrary.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method. The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in `Perm`, `Pack`, or `CCS` format.

To use the DFT functions, you should initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms.

For more information about the fast computation of the discrete Fourier transform, see [\[Mit93\]](#), section 8-2, *Fast Computation of the DFT*.

A special set of Intel IPP functions provides the so called “out-of-order” DFT of the complex signal. In this case, the elements in frequency domain for both forward and inverse transforms can be re-ordered to speed-up the computation of the transforms. This re-ordering is hidden from the user and can be different in different implementations of the functions. However, reversibility of each pair of functions for forward/inverse transforms is ensured.

DFTInitAlloc_R, DFTInitAlloc_C

Initializes the DFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal.

```
ippStatus ippdFTInitAlloc_R_16s(IppdFTSpec_R_16s** pDFTSpec, int length, int
    flag, IppHintAlgorithm hint);
ippStatus ippdFTInitAlloc_R_32f(IppdFTSpec_R_32f** pDFTSpec, int length, int
    flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_R_64f(IppsDFTSpec_R_64f** pDFTSpec, int length, int
    flag, IppHintAlgorithm hint);
```

Case 2: Operation on complex signal.

```
IppStatus ippsDFTInitAlloc_C_16s(IppsDFTSpec_C_16s** pDFTSpec, int length, int
    flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_C_32f(IppsDFTSpec_C_32f** pDFTSpec, int length, int
    flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_C_64f(IppsDFTSpec_C_64f** pDFTSpec, int length, int
    flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_C_16sc(IppsDFTSpec_C_16sc** pDFTSpec, int length,
    int flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_C_32fc(IppsDFTSpec_C_32fc** pDFTSpec, int length,
    int flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_C_64fc(IppsDFTSpec_C_64fc** pDFTSpec, int length,
    int flag, IppHintAlgorithm hint);
```

Parameters

<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in “ Flag and Hint Arguments .”
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments .”
<i>length</i>	Length of the DFT transform.
<i>pDFTSpec</i>	Pointer to the DFT specification structure to be created.

Description

The functions `ippsDFTInitAlloc_R` and `ippsDFTInitAlloc_C` are declared in the `ipps.h` file. These functions create and initialize the DFT specification structure *pDFTSpec* with the following parameters: the transform *length*, the normalization *flag*, and the specific code *hint*. The *length* argument defines the transform length.

ippsDFTInitAlloc_R. The function `ippsDFTInitAlloc_R` initializes the real DFT specification structure.

ippsDFTInitAlloc_C. The function `ippsDFTInitAlloc_C` initializes the complex DFT specification structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDFTSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>length</i> is less than or equal to 0.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

DFTFree_R, DFTFree_C

Closes the DFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal.

```
IppStatus ippDFTFree_R_16s(IppsDFTSpec_R_16s* pDFTSpec);
IppStatus ippDFTFree_R_32f(IppsDFTSpec_R_32f* pDFTSpec);
IppStatus ippDFTFree_R_64f(IppsDFTSpec_R_64f* pDFTSpec);
```

Case 2: Operation on complex signal.

```
IppStatus ippDFTFree_C_16s(IppsDFTSpec_C_16s* pDFTSpec);
IppStatus ippDFTFree_C_32f(IppsDFTSpec_C_32f* pDFTSpec);
IppStatus ippDFTFree_C_64f(IppsDFTSpec_C_64f* pDFTSpec);

IppStatus ippDFTFree_C_16sc(IppsDFTSpec_C_16sc* pDFTSpec);
IppStatus ippDFTFree_C_32fc(IppsDFTSpec_C_32fc* pDFTSpec);
IppStatus ippDFTFree_C_64fc(IppsDFTSpec_C_64fc* pDFTSpec);
```

Parameters

pDFTSpec Pointer to the DFT specification structure to be closed.

Description

The function `ippsDFTFree` is declared in the `ipps.h` file. This function closes the DFT specification structure `pDFTSpec` by freeing all memory associated with the specification created by [ippsDFTInitAlloc_C](#) or [ippsDFTInitAlloc_R](#). Call `ippsDFTFree` after the transform is completed.

ippsDFTFree_R. The function `ippsDFTFree_R` closes the real DFT specification structure.

ippsDFTFree_C. The function `ippsDFTFree_C` closes the complex DFT specification structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDFTSpec</code> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <code>pDFTSpec</code> is incorrect.

DFTGetBufSize_R, DFTGetBufSize_C

Computes the size of the DFT work buffer.

Syntax

Case 1: Operation on real signal.

```
IppStatus ippsDFTGetBufSize_R_16s(const IppsDFTSpec_R_16s* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_R_32f(const IppsDFTSpec_R_32f* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_R_64f(const IppsDFTSpec_R_64f* pDFTSpec, int* pSize);
```

Case 2: Operation on complex signal.

```
IppStatus ippsDFTGetBufSize_C_16s(const IppsDFTSpec_C_16s* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_C_32f(const IppsDFTSpec_C_32f* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_C_64f(const IppsDFTSpec_C_64f* pDFTSpec, int* pSize);

IppStatus ippsDFTGetBufSize_C_16sc(const IppsDFTSpec_C_16sc* pDFTSpec, int* pSize);
```

```
IppStatus ippsDFTGetBufSize_C_32fc(const IppsDFTSpec_C_32fc* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_C_64fc(const IppsDFTSpec_C_64fc* pDFTSpec, int* pSize);
```

Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSize</i>	Pointer to the DFT work buffer size value.

Description

The functions `ippsDFTGetBufSize_R` and `ippsDFTGetBufSize_C` are declared in the `ipps.h` file. These functions compute the size in bytes of an external memory buffer for the DFT described by the specification structure *pDFTSpec* and stores it in *pSize*.

To use external buffering, call this function after *pDFTSpec* initialization.

ippsDFTGetBufSize_R. The function `ippsDFTGetBufSize_R` gets the size of the real DFT work buffer.

ippsDFTGetBufSize_C. The function `ippsDFTGetBufSize_C` gets the size of the complex DFT work buffer.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDFTSpec</i> is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.

DFTFwd_CToC

Computes the forward discrete Fourier transform of a complex signal.

Syntax

Case 1: Operation on real data type.

```
IppStatus ippsDFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f* pDFTSpec, Ipp8u*
    pBuffer);

IppStatus ippsDFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f* pDFTSpec, Ipp8u*
    pBuffer);

IppStatus ippsDFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s* pDFTSpec, int
    scaleFactor, Ipp8u* pBuffer);
```

Case 2: Operation on complex data type.

```
IppStatus ippsDFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
    IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSrc</i>	Pointer to the input array containing complex values.
<i>pDst</i>	Pointer to the output array containing complex values.
<i>pSrcRe</i>	Pointer to the input array containing real parts of the signal.
<i>pSrcIm</i>	Pointer to the input array containing imaginary parts of the signal.
<i>pDstRe</i>	Pointer to the output array containing real parts of the signal.
<i>pDstIm</i>	Pointer to the output array containing imaginary parts of the signal.
<i>pBuffer</i>	Pointer to the work buffer.

scaleFactor Refer to [Integer Scaling](#) in Chapter 2.

Description

The function `ippsDFTFwd_CToC` is declared in the `ipps.h` file. This function computes the forward DFT of a complex signal according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, e.g., with `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the corresponding function [ippsDFTGetBufSize_C](#) prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi \frac{kn}{N}\right),$$

where k is the index of elements in the frequency domain, n is the index of elements in the time domain, N is the input signal *length*, and A is a multiplier defined by *flag*. Also, $x(n)$ is *pSrc[n]* and $X(k)$ is *pDst[k]*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

DFTInv_CToC

Computes the inverse discrete Fourier transform of a complex signal.

Syntax

Case 1: Operation on real data type.

```
IppStatus ippsDFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f* pDFTSpec, Ipp8u*
    pBuffer);

IppStatus ippsDFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f* pDFTSpec, Ipp8u*
    pBuffer);

IppStatus ippsDFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s* pDFTSpec, int
    scaleFactor, Ipp8u* pBuffer);
```

Case 2: Operation on complex data type.

```
IppStatus ippsDFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    const IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
    IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSrc</i>	Pointer to the input array containing complex values.
<i>pDst</i>	Pointer to the output array containing complex values.
<i>pSrcRe</i>	Pointer to the input array containing real parts of the signal.
<i>pSrcIm</i>	Pointer to the input array containing imaginary parts of the signal.
<i>pDstRe</i>	Pointer to the output array containing real parts of the signal.
<i>pDstIm</i>	Pointer to the output array containing imaginary parts of the signal.
<i>pBuffer</i>	Pointer to the work buffer.

scaleFactor Refer to [Integer Scaling](#) in Chapter 2.

Description

The function `ippsDFTInv_CToC` is declared in the `ipps.h` file. This function computes the inverse DFT of a complex signal according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, e.g., with `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the corresponding function [ippsDFTGetBufSize_C](#) prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The inverse DFT functionality can be described as follows:

$$x(n) = B \sum_{k=0}^{N-1} X(k) \cdot \exp\left(j2\pi \frac{kn}{N}\right),$$

where k is the index of elements in the frequency domain, n is the index of elements in the time domain, N is the input signal *length*, and B is a multiplier defined by *flag*. Also, $x(n)$ is *pDst[n]* and $X(k)$ is *pSrc[k]*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

DFTFwd_RToPack, DFTFwd_RToPerm, DFTFwd_RToCCS

Computes the forward discrete Fourier transform of a real signal.

Syntax

Case 1: Result in Pack format.

```
IppStatus ippsDFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 2: Result in Perm format.

```
IppStatus ippsDFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: Result in CCS format.

```
IppStatus ippsDFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSrc</i>	Pointer to the input array containing real values .

<i>pDst</i>	Pointer to the output array containing packed complex values.
<i>pBuffer</i>	Pointer to the work buffer.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDFTFwd` is declared in the `ipps.h` file. It compute the forward DFT of a real signal according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

The result of the forward transform (i.e. in the frequency-domain) of real signals is represented in several possible packed formats: [Pack](#), [Perm](#), or [CCS](#). The data can be packed due to the symmetry property of the DFT transform of a real signal. [Table 7-4](#) and [Table 7-5](#) show how the output result is represented in each of the packed format.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the corresponding function [ippsDFTGetBufSize_R](#) prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi \frac{kn}{N}\right),$$

where k is the index of elements in the frequency domain, n is the index of elements in the time domain, N is the input signal *length*, and A is a multiplier defined by *flag*. Also, $x(n)$ is `pSrc[n]` and $X(k)$ is `pDst[k]`.

ippsDFTFwd_PackToR. This function computes the forward DFT and stores the result in `Pack` format.

ippsDFTFwd_PermToR. This function computes the forward DFT and stores the result in `Perm` format.

ippsDFTFwd_RTtoCCS. This function computes the forward DFT and stores the result in CCS format.

[Example 7-6](#) shows how to initialize the specification and call `ippsDFTFwd_RTtoCCS_32f`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

Example 7-6 Using the `ippsDFTFwd_RTtoCCS` Function

```
IppStatus dft( void ) {
    Ipp32f x[7], X[8];
    int n;
    IppStatus status;
    IppsDFTSpec_R_32f* spec;
    status = ippsDFTInitAlloc_R_32f(&spec, 7, IPP_FFT_DIV_INV_BY_N,
        ippAlgHintNone);
    for( n=0; n<7; ++n ) x[n] = (float) cos(IPP_2PI * n * 14 / 49);
    status = ippsDFTFwd_RTtoCCS_32f( x, X, spec, NULL );
    ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
    ippsDFTFree_R_32f( spec );
    printf_32f("dft magn =", x, 4, status );
    return status;
}
```

Output:

```
dft magn = 0.000000 0.000000 3.500000 0.000000
```

Matlab* analog:

```
>> N=7;F=14/49;n=0:N-1;x=cos(2*pi*n*F);y=abs(fft(x));y(1:4)
```

DFTInv_PackToR, DFTInv_PermToR, DFTInv_CCSToR

Computes the inverse discrete Fourier transform of a real signal.

Syntax

Case 1: Input data in Pack format.

```
IppStatus ippsDFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 2: Input data in Perm format.

```
IppStatus ippsDFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: Input data in CSS format.

```
IppStatus ippsDFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSrc</i>	Pointer to the input array containing packed complex values.

<i>pDst</i>	Pointer to the output array containing real values.
<i>pBuffer</i>	Pointer to the work buffer.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The `ippsDFTInv` function is declared in the `ipps.h` file. They compute the inverse DFT according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

The input data (i.e. in the frequency-domain) are represented in several possible packed formats: [Pack](#), [Perm](#), or [CCS](#). [Table 7-4](#) and [Table 7-5](#) show how the input complex data can be represented in each of the packed format.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the corresponding function [ippsDFTGetBufSize_R](#) prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The inverse DFT functionality can be described as follows:

$$x(n) = B \sum_{k=0}^{N-1} X(k) \cdot \exp\left(j2\pi \frac{kn}{N}\right),$$

where k is the index of elements in the frequency domain, n is the index of elements in the time domain, N is the input signal *length*, and B is a multiplier defined by *flag*. Also, $x(n)$ is `pDst[n]` and $X(k)$ is `pSrc[k]`.

ippsDFTInv_PackToR. This function computes the inverse DFT or input data in `Pack` format.

ippsDFTInv_PermToR. This function computes the inverse DFT for input data in `Perm` format.

ippsDFTInv_CCSToR. This function computes the inverse DFT or input data in `CCS` format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

DFTOutOrdInitAlloc_C

Initializes the out-of-order discrete Fourier transform structure.

Syntax

```
IppStatus ippDFTOutOrdInitAlloc_C_32fc(IppsDFTOutOrdSpec_C_32fc** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);
IppStatus ippDFTOutOrdInitAlloc_C_64fc(IppsDFTOutOrdSpec_C_64fc** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);
```

Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure to be created.
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in “ Flag and Hint Arguments .”
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments .”
<i>length</i>	Length of the DFT transform.

Description

The function `ippDFTOutOrdInitAlloc_C` is declared in the `ipps.h` file. This function creates and initializes the specification structure *pDFTSpec* for the [out-of-order DFT](#) with the following parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFFTSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>length</i> is less than or equal to 0.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

DFTOutOrdFree_C

Closes the out-of-order discrete Fourier transform structure.

Syntax

```
IppStatus ippDFTOutOrdFree_C_32fc(IppsDFTOutOrdSpec_C_32fc* pDFTSpec);
IppStatus ippDFTOutOrdFree_C_64fc(IppsDFTOutOrdSpec_C_64fc* pDFTSpec);
```

Parameters

pDFTSpec Pointer to the DFT specification structure to be closed.

Description

The function `ippDFTOutOrdFree` is declared in the `ipps.h` file. This function closes the specification structure *pDFTSpec* for the [out-of-order DFT](#) by freeing all memory associated with the specification created by [ippDFTOutOrdInitAlloc_C](#). Call `ippDFTOutOrdFree` function after the transform is completed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDFTSpec</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.

DFTOutOrdGetBufSize_C

Computes the size of the work buffer for out-of-order discrete Fourier transform.

Syntax

```
IppStatusippsDFTOutOrdGetBufSize_C_32fc(constIppsDFTOutOrdSpec_C_32fc*pDFTSpec,
    int* pSize);
IppStatusippsDFTOutOrdGetBufSize_C_64fc(constIppsDFTOutOrdSpec_C_64fc*pDFTSpec,
    int* pSize);
```

Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSize</i>	Pointer to the DFT work buffer size value.

Description

The function `ippsDFTOutOrdGetBufSize` is declared in the `ipps.h` file. This function computes the size in bytes of an external memory buffer for the [out-of-order DFT](#) described by the specification structure *pDFTSpec*, and stores it in *pSize*.

To use external buffering, call this function after *pDFTSpec* initialization.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDFTSpec</i> is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.

DFTOutOrdFwd_CToC

Computes the forward out-of-order discrete Fourier transform.

Syntax

```
IppStatus ippsDFTOutOrdFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

Parameters

<i>pSrc</i>	Pointer to the source data.
<i>pDst</i>	Pointer to the output data.
<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pBuffer</i>	Pointer to the work buffer.

Description

The function `ippsDFTOutOrdFwd_CToC` is declared in the `ipps.h` file. This function computes the forward out-of-order DFT of a complex signal *pSrc* according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*, and store the result in *pDst*.

This function is analogous to `ippsDFTFwd_CToC` and has the similar functionality. The difference is that the elements in frequency domain *pDst* can be rearranged if this helps to speed up computations. This procedure is hidden from the user and depends on the specific implementation of the functions. However, the reversibility of each pair of functions for forward/inverse transforms is ensured.

The *pBuffer* argument provides the DFT functions with the necessary memory and allows to avoid memory allocation within the functions. The buffer helps to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the function [ippsDFTOutOrdGetBufSize_C](#) prior to using the DFT computation functions. If a NULL pointer is passed, memory will be allocated by the DFT computation functions internally.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

DFTOutOrdInv_CToC

Computes the inverse out-of-order discrete Fourier transform.

Syntax

```
IppStatus ippsDFTOutOrdInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

Parameters

<i>pSrc</i>	Pointer to the source data.
<i>pDst</i>	Pointer to the output data.
<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pBuffer</i>	Pointer to the work buffer.

Description

The function `ippsDFTOutOrdInv_CToC` is declared in the `ipps.h` file. This function computes the inverse out-of-order DFT of a complex signal *pSrc* according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*, and store the result in *pDst*.

This function is analogous to `ippuDFTInv_CToC` and has the similar functionality. The difference is that the elements in frequency domain `pSrc` can be rearranged if this helps to speed up computations. This procedure is hidden from the user and depends on the specific implementation of the functions. However, the reversibility of each pair of functions for forward/inverse transforms is ensured.

The `pBuffer` argument provides the DFT functions with the necessary memory and allows to avoid memory allocation within the functions. The buffer helps to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the function [ippuDFTOutOrdGetBufSize_C](#) prior to using the DFT computation functions. If a NULL pointer is passed, memory will be allocated by the DFT computation functions internally.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <code>pDFTSpec</code> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

DFT for a Given Frequency (Goertzel) Functions

The functions described in this section compute a single or a number of the discrete Fourier transforms for a given frequency. Note that the DFT exists only for the following normalized frequencies: $0, 1/N, 2/N, \dots, (N-1)/N$, where N is the number of time domain samples. Therefore you must select the frequency value from the above set.

These Intel IPP functions use a Goertzel algorithm [[Mit98](#)] and are more efficient when a small number of DFT values is needed.

Some of the functions compute two values, not one. The applications computing several values, e.g., the dual-tone multi frequency signal detection, work faster, especially on Intel® processors with SIMD instructions.

Goertz

Computes the discrete Fourier transform for a given frequency for a single complex signal.

Syntax

```
IppStatus ippsGoertz_32f(const Ipp32f* pSrc, int len, Ipp32fc* pVal, Ipp32f
    rFreq);
IppStatus ippsGoertz_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pVal, Ipp32f
    rFreq);
IppStatus ippsGoertz_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pVal, Ipp64f
    rFreq);
IppStatus ippsGoertz_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16sc* pVal, Ipp32f
    rFreq, int scaleFactor);
IppStatus ippsGoertz_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pVal,
    Ipp32f rFreq, int scaleFactor);
IppStatus ippsGoertzQ15_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pVal,
    Ipp16s rFreqQ15, int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the input complex data vector.
<i>len</i>	Number of elements in the vector.
<i>pVal</i>	Pointer to the output DFT value.
<i>rFreq</i>	Single relative frequency value [0, 1.0).
<i>rFreqQ15</i>	Single relative frequency value in Q15 format [0, 32767].
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsGoertz` is declared in the `ipps.h` file. This function computes a DFT for a complex input `len`-length signal `pSrc` for a given frequency `rFreq`, and stores the result in `pVal`.

ippsGoertzQ15. This function operates with relative frequency in Q15 format. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 1.0).

The functionality of the Goertzel algorithm can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi \frac{kn}{N}\right),$$

where k/N is the normalized *rFreq* value for which the DFT is computed.

[Example 7-7](#) illustrates the use of Goertzel functions for selecting the magnitudes of a given frequency when computing DFTs.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsRelFreqErr</code>	Indicates an error when <i>rFreq</i> is out of range.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Example 7-7 Using Goertzel Functions for Selecting Magnitudes of a Given Frequency

```

IppStatus goertzel( void ) {
#undef  LEN
#define LEN 100
    IppStatus status;
    Ipp32fc *x = ippsMalloc_32fc( LEN ), y;
    int n;
    ///generate a signal of 60 Hz freq that
    /// is sampled with 400 Hz freq
    for( n=0; n<LEN; ++n) {
        x[n].re =(Ipp32f)sin(IPP_2PI * n * 60 / 400);
        x[n].im = 0;
    }
    status = ippsGoertz_32fc( x, LEN, &y, 60.0f / 400 );
    printf_32fc("goertz =", &y, 1, status );
    ippsFree( x );
    return status;
}

```

Output:

```

goertz = {0.000090,-50.000008}
Matlab* Analog
>> N=100;F=60/400;n=0:N-1;x=sin(2*pi*n*F);y=fft(x);n=N*F;y(n+1)

```

GoertzTwo

Computes two discrete Fourier transforms for a given frequency for a single complex signal.

Syntax

```
IppStatus ippsGoertzTwo_32fc(const Ipp32fc* pSrc, int len, Ipp32fc pVal[2],
    const Ipp32f rFreq[2]);
IppStatus ippsGoertzTwo_64fc(const Ipp64fc* pSrc, int len, Ipp64fc pVal[2],
    const Ipp64f rFreq[2]);
IppStatus ippsGoertzTwo_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc pVal[2],
    const Ipp32f rFreq[2], int scaleFactor);
IppStatus ippsGoertzTwoQ15_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc
    pVal[2], const Ipp16s rFreqQ15[2], int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the input complex data vector.
<i>len</i>	Number of elements in the vector.
<i>pVal</i>	Pointer to the output DFT values.
<i>rFreq</i>	Array of two relative frequency values [0, 1.0).
<i>rFreqQ15</i>	Array of two relative frequency values in Q15 format [0, 32767].
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsGoertzTwo` is declared in the `ipps.h` file. This function computes two DFTs for a complex input *len*-length signal *pSrc* for two given frequencies *rFreq*, and stores the result in *pVal*. The computation of two DFTs on an Intel® Pentium® III processor is performed at the same speed as one. Therefore, the applications computing several DFTs are faster.

The functionality of the Goertzel algorithm can be described as follows:

$$Y(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi \frac{kn}{N}\right),$$

where k/N is one of the normalized *rFreq* values for which the DFTs are computed.

ippsGoertzTwoQ15. This function operates with relative frequencies in Q15 format. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 1.0).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are NULL.
<code>ippStsRelFreqErr</code>	Indicates an error when <i>rFreq</i> is out of range.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.

Discrete Cosine Transform Functions

This section describes the functions that compute the discrete cosine transform of a signal. DCT functions used in the Intel IPP signal processing data-domain implement the modified computation algorithm proposed in [[Rao90](#)].

DCTFwdInitAlloc, DCTInvInitAlloc

Initializes the discrete cosine transform structure.

Syntax

Case 1: Forward transform.

```
IppStatus ippsDCTFwdInitAlloc_16s(IppsDCTFwdSpec_16s** pDCTSpec, int length,
    IppHintAlgorithm hint);
IppStatus ippsDCTFwdInitAlloc_32f(IppsDCTFwdSpec_32f** pDCTSpec, int length,
    IppHintAlgorithm hint);
IppStatus ippsDCTFwdInitAlloc_64f(IppsDCTFwdSpec_64f** pDCTSpec, int length,
    IppHintAlgorithm hint);
```

Case 2: Inverse transform.

```
IppStatus ippsDCTInvInitAlloc_16s(IppsDCTInvSpec_16s** pDCTSpec, int length,
    IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_32f(IppsDCTInvSpec_32f** pDCTSpec, int length,
    IppHintAlgorithm hint);
```

```
IppStatus ippsDCTInvInitAlloc_64f(IppsDCTInvSpec_64f** pDCTSpec, int length,
    IppHintAlgorithm hint);
```

Parameters

<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in “ Flag and Hint Arguments .”
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments .”
<i>length</i>	Number of samples in the DCT.
<i>pDCTSpec</i>	Pointer to the DCT specification structure to be created.

Description

The functions `ippsDCTFwdInitAlloc` and `ippsDCTInvInitAlloc` are declared in the `ipps.h` file. These functions create and initialize the DCT specification structure *pDCTSpec* with the following parameters: the transform *length*, the normalization *flag*, and the specific code *hint*. The *length* argument defines the transform length.

ippsDCTFwdInitAlloc. The function `ippsDCTFwdInitAlloc` initializes the forward DCT specification structure.

ippsDCTInvInitAlloc. The function `ippsDCTInvInitAlloc` initializes the inverse DCT specification structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDCTSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>length</i> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

DCTFwdFree, DCTInvFree

Closes a discrete cosine transform structure.

Syntax

Case 1: Forward transform.

```
IppStatus ippDCTFwdFree_16s(IppsDCTFwdSpec_16s* pDCTSpec);
IppStatus ippDCTFwdFree_32f(IppsDCTFwdSpec_32f* pDCTSpec);
IppStatus ippDCTFwdFree_64f(IppsDCTFwdSpec_64f* pDCTSpec);
```

Case 2: Inverse transform.

```
IppStatus ippDCTInvFree_16s(IppsDCTInvSpec_16s* pDCTSpec);
IppStatus ippDCTInvFree_32f(IppsDCTInvSpec_32f* pDCTSpec);
IppStatus ippDCTInvFree_64f(IppsDCTInvSpec_64f* pDCTSpec);
```

Parameters

pDCTSpec Pointer to the DCT specification structure to be closed.

Description

The functions `ippDCTFwdFree` and `ippDCTInvFree` are declared in the `ipp.h` file. These functions close the DCT specification structure *pDCTSpec* by freeing all memory associated with the specification created by [ippDCTFwdInitAlloc](#) or [ippDCTInvInitAlloc](#). Call either `ippDCTFwdFree` or `ippDCTInvFree` after the transform is completed.

ippDCTFwdFree. The function `ippDCTFwdFree` closes the forward DCT specification structure.

ippDCTInvFree. The function `ippDCTInvFree` closes the inverse DCT specification structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDCTSpec</i> pointer is NULL.

<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDCTSpec</i> is incorrect.
------------------------------------	--

DCTFwdGetBufSize, DCTInvGetBufSize

Computes the size of the DCT work buffer.

Syntax

Case 1: Forward transform.

```
IppStatus ippdDCTFwdGetBufSize_32f(const IppsDCTFwdSpec_32f* pDCTSpec, int* pSize);
IppStatus ippdDCTFwdGetBufSize_16s(const IppsDCTFwdSpec_16s* pDCTSpec, int* pSize);
IppStatus ippdDCTFwdGetBufSize_64f(const IppsDCTFwdSpec_64f* pDCTSpec, int* pSize);
```

Case 2: Inverse transform.

```
IppStatus ippdDCTInvGetBufSize_16s(const IppsDCTInvSpec_16s* pDCTSpec, int* pSize);
IppStatus ippdDCTInvGetBufSize_32f(const IppsDCTInvSpec_32f* pDCTSpec, int* pSize);
IppStatus ippdDCTInvGetBufSize_64f(const IppsDCTInvSpec_64f* pDCTSpec, int* pSize);
```

Parameters

<i>pDCTSpec</i>	Pointer to the DCT specification structure.
<i>pSize</i>	Pointer to the DCT work buffer size value.

Description

The functions `ippdDCTFwdGetBufSize` and `ippdDCTInvGetBufSize` are declared in the `ipps.h` file. These functions get the work buffer size of the DCT described by the specification structure *pDCTSpec* in bytes and stores in *pSize*.

ippdDCTFwdGetBufSize. The function `ippdDCTFwdGetBufSize` gets the work buffer size of the forward DCT.

ippdDCTInvGetBufSize. The function `ippdDCTInvGetBufSize` gets the work buffer size of the inverse DCT.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDCTSpec</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDCTSpec</i> is incorrect.

DCTFwd, DCTInv

Computes the forward or inverse discrete cosine transform of a signal.

Syntax

Case 1: Forward transform.

```
IppStatus ippSDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDCTFwdSpec_32f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippSDCTFwd_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDCTFwdSpec_64f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippSDCTFwd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDCTFwdSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 2: Inverse transform.

```
IppStatus ippSDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDCTInvSpec_32f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippSDCTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDCTInvSpec_64f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippSDCTInv_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDCTInvSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

<i>pDCTSpec</i>	Pointer to the DCT specification structure.
<i>pSrc</i>	Pointer to the input data array.
<i>pDst</i>	Pointer to the output data array.
<i>pBuffer</i>	Pointer to the DCT work buffer

scaleFactor

Refer to [Integer Scaling](#) in Chapter 2.

Description

The functions `ippsDCTFwd` and `ippsDCTInv` are declared in the `ipps.h` file. These functions compute the forward and inverse discrete cosine transform (DCT). If *length* is a power of 2, the functions use an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *length*, these functions use the direct formulas given below; however, the symmetry of the cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT, $N = \text{length}$,

$$C(k) = \frac{1}{\sqrt{N}} \text{ for } k = 0, \quad C(k) = \frac{\sqrt{2}}{\sqrt{N}} \text{ for } k > 0;$$

$x(n)$ is `pSrc[n]` and $y(k)$ is `pDst[k]` for the forward DCT;

$x(n)$ is `pDst[n]` and $y(k)$ is `pSrc[k]` for the inverse DCT.

The forward DCT is defined by the formula:

$$y(k) = C(k) \sum_{n=0}^{N-1} x(n) \cdot \cos \frac{(2n+1)\pi k}{2N}$$

The definition of the inverse discrete cosine transform is:

$$x(n) = \sum_{k=0}^{N-1} C(k) y(k) \cdot \cos \frac{(2n+1)\pi k}{2N}$$

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained. The *pBuffer* argument provides the DCT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer may also increase up performance if the DCT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

ippsDCTFwd. The function `ippsDCTFwd` computes a forward DCT.

ippsDCTInv. The function `ippsDCTInv` computes an inverse DCT.

[Example 7-8](#) shows how to use the functions `ippsDCTFwd_32f` and `ippsDCTInv_32f` to compress and reconstruct a signal.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDCTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

Example 7-8 Using `ippsDCTFwd` and `ippsDCTInv` to Compress and Reconstruct a Signal

```
void dct( void ) {
#define LEN 256
    Ipp32f x[LEN], y[LEN];
    int n;
    IppsDCTFwdSpec_32f* fspec;
    IppsDCTInvSpec_32f* ispec;
    IppStatus status;
    /// data: Gaussian function, magn =1 and sigma=N/3
    for(n=0; n<LEN; ++n)
        x[n] = (float)(exp(-0.5*(LEN/2-n)*(LEN/2-n)/(LEN/3.0)));
    /// get cosine transform coefficients
    status = ippsDCTFwdInitAlloc_32f( &fspec, LEN, ippAlgHintNone );
    status = ippsDCTFwd_32f( x, y, fspec, NULL );
    ippsDCTFwdFree_32f( fspec );
    /// Set 3/4 of these coefficients to zero
    for(n=LEN/4; n<LEN; ++n) y[n] = 0.0f;
    /// restore signal using len/4 values
    status = ippsDCTInvInitAlloc_32f( &ispec, LEN, ippAlgHintNone );
    status = ippsDCTInv_32f( y, x, ispec, NULL );
    ippsDCTInvFree_32f( ispec );
}
```

Hilbert Transform Functions

The functions described in this section compute a discrete-time analytic signal from a real data sequence using the Hilbert transform. The analytic signal is a complex signal whose real part is a replica of the original data, and imaginary part contains the Hilbert transform. That is, the imaginary part is a version of the original real data with a 90 degrees phase shift. The Hilbert transformed data have the same amplitude and frequency content as the original real data, plus the additional phase information.

HilbertInitAlloc

Initializes the Hilbert transform structure.

Syntax

```
IppStatusippsHilbertInitAlloc_32f32fc(IppsHilbertSpec_32f32fc**pSpec, intlength,
    IppHintAlgorithm hint);
IppStatusippsHilbertInitAlloc_16s32fc(IppsHilbertSpec_16s32fc**pSpec, intlength,
    IppHintAlgorithm hint);
IppStatusippsHilbertInitAlloc_16s16sc(IppsHilbertSpec_16s16sc**pSpec, intlength,
    IppHintAlgorithm hint);
```

Parameters

<i>pSpec</i>	Pointer to pointer to the Hilbert context structure being initialized.
<i>length</i>	Number of samples in the Hilbert transform.
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in “ Flag and Hint Arguments .”

Description

The function `ippsHilbertInitAlloc` is declared in the `ipps.h` file. This function creates and initializes the Hilbert specification structure *pSpec* with the following parameters: the length of the transform *length*, and the specific code indicator *hint*. Call this function before actually using the Hilbert transform function `ippsHilbert`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>length</i> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

HilbertFree

Closes a Hilbert transform structure.

Syntax

```
IppStatus ippHilbertFree_32f32fc(IppsHilbertSpec_32f32fc* pSpec);
IppStatus ippHilbertFree_16s32fc(IppsHilbertSpec_16s32fc* pSpec);
IppStatus ippHilbertFree_16s16sc(IppsHilbertSpec_16s16sc* pSpec);
```

Parameters

pSpec Pointer to the Hilbert specification structure to be closed.

Description

The function `ippHilbertFree` is declared in the `ipps.h` file. This function closes the Hilbert specification structure *pSpec* by freeing all memory associated with the specification created by the function [ippHilbertInitAlloc](#). Call `ippHilbertFree` after the transform is completed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSpec</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pSpec</i> is incorrect.

Hilbert

Computes an analytic signal using the Hilbert transform.

Syntax

```
IppStatus ippsHilbert_32f32fc(const Ipp32f* pSrc, Ipp32fc* pDst,
    IppsHilbertSpec_32f32fc* pSpec);
IppStatus ippsHilbert_16s32fc(const Ipp16s* pSrc, Ipp32fc* pDst,
    IppsHilbertSpec_16s32fc* pSpec);
IppStatus ippsHilbert_16s16sc_Sfs(const Ipp16s* pSrc, Ipp16sc* pDst,
    IppsHilbertSpec_16s16sc* pSpec, int scaleFactor);
```

Parameters

<i>pSpec</i>	Pointer to the Hilbert specification structure.
<i>pSrc</i>	Pointer to the vector containing original real data.
<i>pDst</i>	Pointer to the output array containing complex data.
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsHilbert` is declared in the `ipps.h` file. This function computes a complex analytic signal *pDst*, which contains the original real signal *pSrc* as its real part and computed Hilbert transform as its imaginary part. The Hilbert transform is performed according to the *pSpec* specification parameters: the number of samples *length*, and the specific code *hint*. The input data is zero-padded or truncated to the size of *length* as appropriate. For integer data types, the output result is scaled according to the *scaleFactor* value, which ensures that the output signal range and precision are retained.

[Example 7-9](#) shows how to initialize the specification structure and use the function `ippsHilbert_32f32fc`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.

<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

Example 7-9 Using Hilbert Functions

```

IppStatus hilbert(void) {
    Ipp32f x[10];
    Ipp32fc y[10];
    int n;
    IppStatus status;
    IppsHilbertSpec_32f32fc* spec;

    status = ippsHilbertInitAlloc_32f32fc(&spec, 10, ippAlgHintNone);
    for(n = 0; n < 9; n++) {
        x[n] = (Ipp32f)cos(IPP_2PI * i * 2 / 9);
    }
    status = ippsHilbert_32f32fc(x, y, spec);
    ippsMagnitude_32fc((Ipp32fc*)y, x, 5);
    ippsHilbertFree_32f32fc(spec);
    printf_32f("hilbert magn =", x, 5, status);
    return status;
}

```

Output:

```
hilbert magn = 1.0944 1.1214 1.0413 0.9707 0.9839
```

Matlab* Analog:

```
>> n=0:9; x=cos(2*pi*n*2/9); y=abs(hilbert(x)); y(1:5)
```

Wavelet Transform Functions

This section describes the wavelet transform functions implemented in Intel IPP.

In signal processing, signals can be represented in both frequency and time-frequency domains. In many cases the wavelet transforms become an alternative to short time Fourier transforms.

The discrete wavelet signal can be considered as a set of the coefficients $a_{i,k}$ with two indices, one of which is a “frequency” characteristic and the other is a time localization. The coefficient value corresponds to the localized wave amplitude or to one of basis transform functions. The “frequency” index shows the time scale of the localized wave. Function bases originated from one local wave by decreasing the wave by 2^n in time are the most widely used. Such transforms can be used for building very efficient implementations called fast wavelet transforms by analogy with fast Fourier transforms. [Figure 7-1](#) shows how the time and frequency plane is divided into areas that correspond to the local wave amplitudes. This kind of transforms is implemented in the Intel IPP and referred to as the discrete wavelet transform (DWT).

The DWT is one of the wavelet analysis methods that stem from the basis functions related to the scale factor 2. Thus, there is a basic common element shared by the DWT and the other packet analysis methods.

Likewise another basic element for signal reconstruction or synthesis can be defined, called the one-level inverse DWT. [Figure 7-2](#) shows the diagram of the forward DWT which allows to switch to time-frequency representation shown in [Figure 7-1](#). The diagram includes three levels of decomposition. [Figure 7-3](#) shows the corresponding procedure of signal reconstruction based on the elementary one-level inverse transform.

The implementation of discrete multi-scale transforms is based on the use of interpolation and decimation filters with the resampling factor 2. The basis of the multi-scale signal decomposition and reconstruction functions uniquely defines the filter parameters. The Intel IPP multi-scale transform functions use filters with finite impulse response.

The Primitives contains two sets of functions.

- Transforms designed for fixed filter banks. These transforms yield the highest performance.
- Transforms that enable the user to work with arbitrary filters. These functions use effective polyphase filtration algorithms. The transform interface gives the option of processing the data in blocks, including in real-time applications.

Figure 7-1 Wavelet Decomposition Coefficients in Time-Frequency Domain

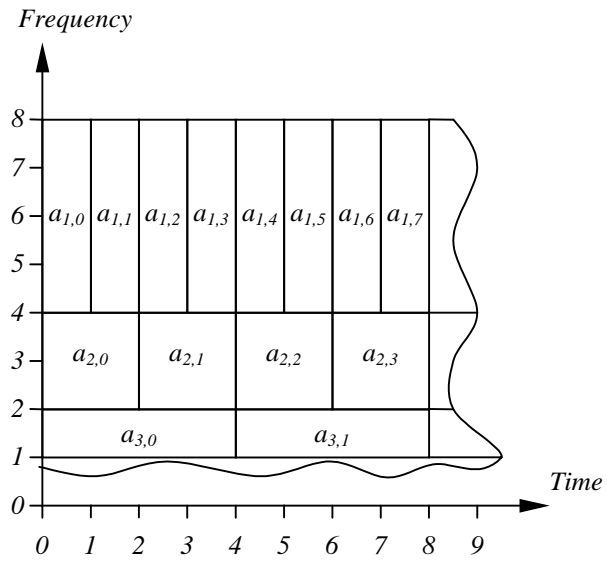


Figure 7-2 Three-Level Discrete Wavelet Decomposition

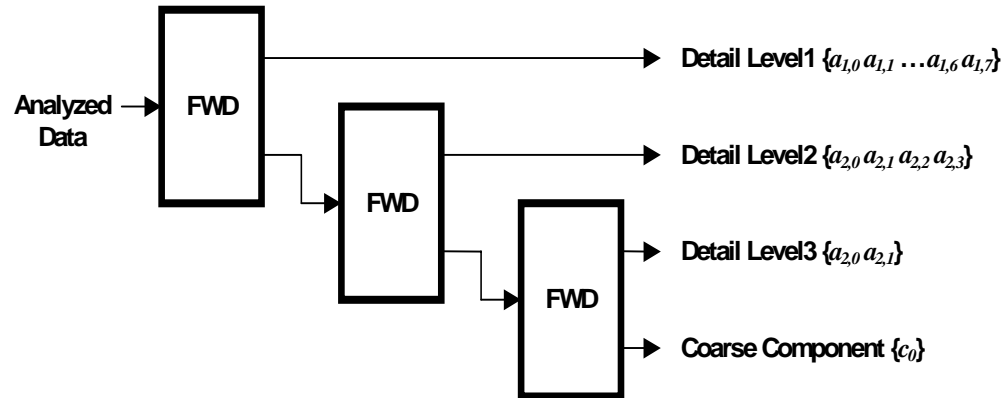
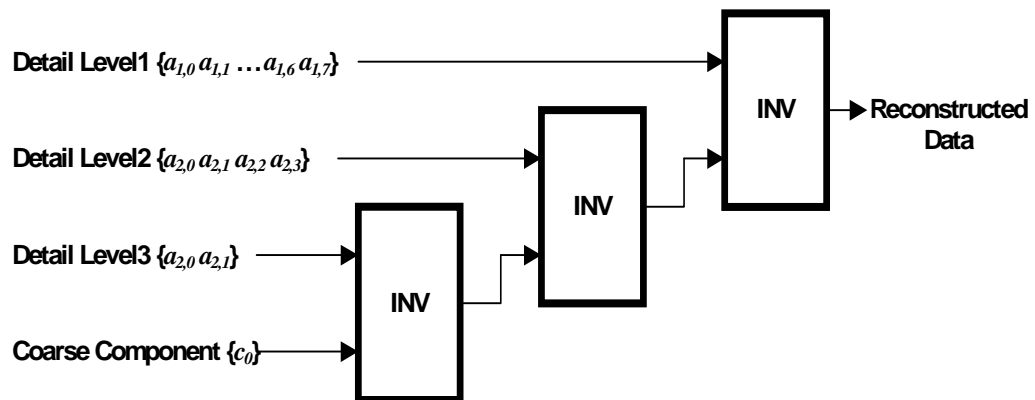


Figure 7-3 Three-Level Discrete Wavelet Reconstruction



Transforms for Fixed Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for fixed filter banks.

WTHaarFwd, WTHaarInv

Performs forward or inverse single-level discrete wavelet Haar transforms.

Syntax

Case 1: Forward transform.

```
IppStatus ippsWTHaarFwd_8s(const Ipp8s* pSrc, int lenSrc, Ipp8s* pDstLow,
    Ipp8s* pDstHigh);
IppStatus ippsWTHaarFwd_16s(const Ipp16s* pSrc, int lenSrc, Ipp16s* pDstLow,
    Ipp16s* pDstHigh);
IppStatus ippsWTHaarFwd_32s(const Ipp32s* pSrc, int lenSrc, Ipp32s* pDstLow,
    Ipp32s* pDstHigh);
IppStatus ippsWTHaarFwd_64s(const Ipp64s* pSrc, int lenSrc, Ipp64s* pDstLow,
    Ipp64s* pDstHigh);
IppStatus ippsWTHaarFwd_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f* pDstLow,
    Ipp32f* pDstHigh);
IppStatus ippsWTHaarFwd_64f(const Ipp64f* pSrc, int lenSrc, Ipp64f* pDstLow,
    Ipp64f* pDstHigh);

IppStatus ippsWTHaarFwd_8s_Sfs(const Ipp8s* pSrc, int lenSrc, Ipp8s* pDstLow,
    Ipp8s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_16s_Sfs(const Ipp16s* pSrc, int lenSrc, Ipp16s*
    pDstLow, Ipp16s* pDstHigh, int scaleFactor );
IppStatus ippsWTHaarFwd_32s_Sfs(const Ipp32s* pSrc, int lenSrc, Ipp32s*
    pDstLow, Ipp32s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_64s_Sfs(const Ipp64s* pSrc, int lenSrc, Ipp64s*
    pDstLow, Ipp64s* pDstHigh, int scaleFactor);
```

Case 2: Inverse transform.

```

IppStatus ippsWTHaarInv_8s(const Ipp8s* pSrcLow, const Ipp8s* pSrcHigh, Ipp8s*
    pDst, int lenDst);

IppStatus ippsWTHaarInv_16s(const Ipp16s* pSrcLow, const Ipp16s* pSrcHigh,
    Ipp16s* pDst, int lenDst);

IppStatus ippsWTHaarInv_32s(const Ipp32s* pSrcLow, const Ipp32s* pSrcHigh,
    Ipp32s* pDst, int lenDst);

IppStatus ippsWTHaarInv_64s(const Ipp64s* pSrcLow, const Ipp64s* pSrcHigh,
    Ipp64s* pDst, int lenDst);

IppStatus ippsWTHaarInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
    Ipp32f* pDst, int lenDst);

IppStatus ippsWTHaarInv_64f(const Ipp64f* pSrcLow, const Ipp64f* pSrcHigh,
    Ipp64f* pDst, int lenDst);

IppStatus ippsWTHaarInv_8s_Sfs(const Ipp8s* pSrcLow, const Ipp8s* pSrcHigh,
    Ipp8s* pDst, int lenDst, int scaleFactor);

IppStatus ippsWTHaarInv_16s_Sfs(const Ipp16s* pSrcLow, const Ipp16s* pSrcHigh,
    Ipp16s* pDst, int lenDst, int scaleFactor);

IppStatus ippsWTHaarInv_32s_Sfs(const Ipp32s* pSrcLow, const Ipp32s* pSrcHigh,
    Ipp32s* pDst, int lenDst, int scaleFactor);

IppStatus ippsWTHaarInv_64s_Sfs(const Ipp64s* pSrcLow, const Ipp64s* pSrcHigh,
    Ipp64s* pDst, int lenDst, int scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the array with the input signal for the ippsWTHaarFwd function.
<i>lenSrc</i>	Number of elements in the source vector <i>pSrc</i> .
<i>pDstLow</i>	Pointer to the array with the coarse “low frequency” components of the output of the ippsWTHaarFwd function.
<i>pDstHigh</i>	Pointer to the array with the detail “high frequency” components of the output of the ippsWTHaarFwd function.
<i>pSrcLow</i>	Pointer to the array with the coarse “low frequency” components of the input to the ippsWTHaarInv function.
<i>pSrcHigh</i>	Pointer to the array with the detail “high frequency” components of the input to the ippsWTHaarInv function.

<i>pDst</i>	Pointer to the array with the the output signal of the <code>ippsWTHaarInv</code> function.
<i>lenDst</i>	Number of elements in the destination vector <i>pDst</i>
<i>scaleFactor</i>	Refer to Integer Scaling in Chapter 2.

Description

The functions `ippsWTHaarFwd` and `ippsWTHaarInv` are declared in the `ipps.h` file. These functions perform forward and inverse single-level discrete Haar transforms. These transforms are orthogonal and reconstruct the original signal perfectly.

The forward transform can be considered as wavelet signal decomposition with lowpass decimation filter coefficients $\{1/2, 1/2\}$ and highpass decimation filter coefficients $\{1/2, -1/2\}$.

The inverse transform is represented as wavelet signal reconstruction with lowpass interpolation filter coefficients $\{1, 1\}$ and highpass interpolation filter coefficients $\{-1, 1\}$.

The decomposition filter coefficients are frequency response normalized to provide the same value range for both input and output signals. Thus, the amplitude of the low pass filter frequency response is 1 for zero-valued frequency, and the amplitude of the high pass filter frequency response is also 1 for the frequency value near to 0.5.

As the absolute values of the interpolation filter coefficients are equal to 1, the reconstruction of the signal requires few operations. It is well suited for usage in data compression applications. As the decomposition filter coefficients are powers of 2, the integer functions perform lossless decomposition with the *scaleFactor* value equal to -1. To avoid saturation, use higher-precision data types.

Note that the filter coefficients can be power spectral response normalized, see [Str96] for more information. Thus, the decomposition filter coefficients are $\{2^{-1/2}, 2^{-1/2}\}$ and $\{2^{-1/2}, -2^{-1/2}\}$; accordingly; the reconstruction filter coefficients are $\{2^{-1/2}, 2^{-1/2}\}$ and $\{-2^{-1/2}, 2^{-1/2}\}$.

In the following definition of the forward single-level discrete Haar transform, $N=lenSrc$. The coarse “low-frequency” component $c(k)$ is `pDstLow[k]` and the detail “high-frequency” component $d(k)$ is `pDstHigh[k]`; also $x(2k)$ and $x(2k+1)$ are even and odd values of the input signal *pSrc*, respectively.

$$c(k) = (x(2k) + x(2k+1)) / 2$$

$$d(k) = (x(2k+1) - x(2k)) / 2$$

In the inverse direction, $N = \text{lenDSt}$. The coarse “low-frequency” component $c(k)$ is $pSrcLow[k]$ and the detail “high-frequency” component $d(k)$ is $pSrcHigh[k]$; also $y(2i)$ and $y(2i+1)$ are even and odd values of the output signal $pDSt$, respectively.

$$y(2i) = c(i) - d(i)$$

$$y(2i+1) = c(i) + d(i)$$

For even length N , $0 \leq i < N/2$ and $0 \leq i < N/2$. Also, “low-frequency” and “high-frequency” components are of size $N/2$ for both original and reconstructed signals. The total length of components is equal to the signal length N .

In case of odd length N , the vector is considered as a vector of the extended length $N+1$ whose two last elements are equal to each other $x[N] = x[N-1]$. The last elements of the coarse and detail components of the decomposed signal are defined as follows:

$$c((N+1)/2-1) = x(N-1)$$

$$d((N+1)/2-1) = 0$$

Correspondingly, the last element of the reconstructed signal is defined as:

$$y(N) = y(N-1) = c((N+1)/2-1)$$

For odd length N , $0 \leq i < (N-1)/2$ and $0 \leq i < (N-1)/2$, assuming that $c((N+1)/2-1) = x(N-1)$ and $y(N-1) = c((N+1)/2-1)$. The “low-frequency” component is of size $(N+1)/2$. The “high-frequency” component is of size $(N-1)/2$, because the last element $d((N+1)/2-1)$ is always equal to 0. The total length of components is also N .

Such an approach applies continuation of boundaries for filters having the symmetry properties, see [\[Brislaw94\]](#).

When performing block mode transforms, take into consideration that for decomposition and reconstruction of even-length signals no extrapolations at the boundaries is used. In case of odd-length signals, a symmetric continuation of the signal boundary with the last point replica is applied.

When it is necessary to have a continuous set of output blocks, all the input blocks are to be of even length, besides the last one (which can be either of odd or even length). Thus, if the whole amount of elements is odd, only the last block can be of odd length.

ippSWTHaarFwd. The function `ippSWTHaarFwd` performs the forward single-level discrete Haar transform of a lenSrc -length signal $pSrc$ and stores the decomposed coarse “low-frequency” components in $pDStLow$, and the detail “high-frequency” components in $pDStHigh$.

ippsWTHaarInv. The function `ippsWTHaarInv` performs the inverse single-level discrete Haar transform of the coarse “low-frequency” components *pSrcLow* and detail “high-frequency” components *pSrcHigh*, and stores the reconstructed signal in the *lenDst* -length vector *pDst*.

[Example 7-10](#) illustrates the use of the function `ippsWTHaarFwd_32f`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than 4 for the function <code>ippsWinBlackmanOpt</code> and less than 3 for all other functions of the family.

Example 7-10 Using the `ippsWTHaarFwd` Function

```

IppStatus wthaar(void) {
    Ipp32f x[8], lo[4], hi[4];
    IppStatus status;
    ippsSet_32f(7, x, 8);  --x[4];
    status = ippsWTHaarFwd_32f(x, 8, lo, hi);
    printf_32f("WT Haar low  =", lo, 4, status);
    printf_32f("WT Haar high =", hi, 4, status);
    return status;
}

```

Output:

```

WT Haar low  =  7.000000 7.000000 6.500000 7.000000
WT Haar high =  0.000000 0.000000 0.500000 0.000000

```

See Also

For more information on wavelet transforms see [\[Strang96\]](#) and [\[Brislaw94\]](#).

Transforms for User Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for user filter banks.

WTFwdInitAlloc, WTInvInitAlloc

Initializes the wavelet transform structure.

Syntax

Case 1: Forward transform.

```
IppStatus ippsWTFwdInitAlloc_32f(IppsWTFwdState_32f** pState, const Ipp32f*
    pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int
    offsHigh);

IppStatus ippsWTFwdInitAlloc_8s32f(IppsWTFwdState_8s32f** pState, const Ipp32f*
    pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int
    offsHigh);

IppStatus ippsWTFwdInitAlloc_8u32f(IppsWTFwdState_8u32f** pState, const Ipp32f*
    pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int
    offsHigh);

IppStatus ippsWTFwdInitAlloc_16s32f(IppsWTFwdState_16s32f** pState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);

IppStatus ippsWTFwdInitAlloc_16u32f(IppsWTFwdState_16u32f** pState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
```

Case 2: Inverse transform.

```
IppStatus ippsWTInvInitAlloc_32f(IppsWTInvState_32f** pState, const Ipp32f*
    pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int
    offsHigh);

IppStatus ippsWTInvInitAlloc_32f8s(IppsWTInvState_32f8s** pState, const Ipp32f*
    pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int
    offsHigh);
```

```
IppStatus ippsWTInvInitAlloc_32f8u(IppsWTInvState_32f8u** pState, const Ipp32f*
    pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int
    offsHigh);

IppStatus ippsWTInvInitAlloc_32f16s(IppsWTInvState_32f16s** pState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);

IppStatus ippsWTInvInitAlloc_32f16u(IppsWTInvState_32f16u** pState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
```

Parameters

<i>pState</i>	Pointer to the location to store the allocated and initialized state structure.
<i>pTapsLow</i>	Pointer to the vector of lowpass filter taps.
<i>lenLow</i>	Number of taps in the lowpass filter.
<i>offsLow</i>	Additional delay (offset) of the lowpass filter.
<i>pTapsHigh</i>	Pointer to the vector of highpass filter taps.
<i>lenHigh</i>	Number of taps in the highpass filter.
<i>offsHigh</i>	Additional delay (offset) of the highpass filter.

Description

The functions `ippsWTFwdInitAlloc` and `ippsWTInvInitAlloc` are declared in the `ipps.h` file. These functions create and initialize the WT state structure *pState* with the following parameters: the lowpass and highpass filter taps *pTapsLow* and *pTapsHigh*, lengths *lenLow* and *lenHigh*, input additional delays *offsLow* and *offsHigh*, respectively.

ippsWTFwdInitAlloc. The function `ippsWTFwdInitAlloc` initializes the forward WT state structure.

ippsWTInvInitAlloc. The function `ippsWTInvInitAlloc` initializes the inverse WT state structure.

Application Notes

These functions initialize the wavelet transform state structure, allocate memory for the state structure, initialize it, and returns the *pState* pointer to the state structure. The initialization procedures are implemented separately for forward and inverse transforms. To perform both

forward and inverse wavelet transforms, create two separate state structures. In general, the meanings of initialization parameters of forward and inverse transforms are similar. Each function has parameters describing of a pair of filters. The forward transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of analysis filters. The inverse transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of synthesis filters. Besides lengths and sets of taps the functions allow to specify an additional delay *offsLow* and *offsHigh* for each filter. The adjustable values of delays allow to synchronize:

- group delays for highpass and lowpass filters;
- delays between data of different levels in multilevel decomposition and reconstruction algorithms.

For more information about using these parameters, see descriptions of the functions [ippsWTFwd](#) and [ippsWTInv](#) in this chapter. The minimum allowed value of the additional delay for the forward transform is -1. For the inverse transform the delay values must be greater or equal to 0. See descriptions of the functions [ippsWTFwd](#) and [ippsWTInv](#) in this chapter for an example showing how to choose additional delay values. The initialization functions copy filter taps into the state structure *pState*. So all the memory referred to with the pointers can be freed or modified after the functions finished operating. In case of the memory shortage, the function sets a zero pointer to the structure.

Boundaries extrapolation. Typically, reversible wavelet transforms of a bounded signal require data extrapolation towards one or both sides. All internal delay lines are set to zero at the stage of initialization. To set a non-zero signal prehistory, call the function [ippsWTFwdSetDlyLine](#). When processed an entire limited data set, data extrapolation may be performed both towards the start and the end of the data vector. For that, the source data and their initial extrapolation are used to form the delay line, the rest of the signal is subdivided into the main block and the signal end. The signal end data and their extrapolation are used to form the last block.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> , <i>pTapsHigh</i> , or <i>pTapsLow</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>lenLow</i> or <i>lenHigh</i> is less than or equal to 0.
<code>ippStsWtOffsetErr</code>	Indicates an error when the filter delay <i>offsLow</i> or <i>offsHigh</i> is less than -1 for the forward transform; and is less than 0 for the inverse transform.

WTFwdFree, WTInvFree

Closes a wavelet transform structure.

Syntax

Case 1: Forward transform.

```
IppStatus ippsWTFwdFree_32f(IppsWTFwdState_32f* pState);
IppStatus ippsWTFwdFree_8s32f(IppsWTFwdState_8s32f* pState);
IppStatus ippsWTFwdFree_8u32f(IppsWTFwdState_8u32f* pState);
IppStatus ippsWTFwdFree_16s32f(IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwdFree_16u32f(IppsWTFwdState_16u32f* pState);
```

Case 2: Inverse transform.

```
IppStatus ippsWTInvFree_32f(IppsWTInvState_32f* pState);
IppStatus ippsWTInvFree_32f8s(IppsWTInvState_32f8s* pState);
IppStatus ippsWTInvFree_32f8u(IppsWTInvState_32f8u* pState);
IppStatus ippsWTInvFree_32f16s(IppsWTInvState_32f16s* pState);
IppStatus ippsWTInvFree_32f16u(IppsWTInvState_32f16u* pState);
```

Parameters

pState Pointer to the state structure to be closed.

Description

The functions `ippsWTFwdFree` and `ippsWTInvFree` are declared in the `ipps.h` file. These functions close the WT state structure *pState* by freeing all the internal memory associated with the state created by [ippsWTFwdInitAlloc](#) or [ippsWTInvInitAlloc](#). Call either `ippsWTFwdFree` or `ippsWTInvFree` after the transform is completed. If the *pState* pointer is NULL, the function performs no operation and returns the `ippStsNullPtrErr` status.

ippsWTFwdFree. The function `ippsWTFwdFree` closes the forward WT state structure.

ippsWTInvFree. The function `ippsWTInvFree` closes the inverse WT state structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pState</i> is NULL.
<code>ippStsStateMatchErr</code>	Indicates an error when the state identifier <i>pState</i> is incorrect.

WTFwd

Computes the forward wavelet transform.

Syntax

```

IppStatus ippsWTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDstLow, Ipp32f* pDstHigh,
    int dstLen, IppsWTFwdState_32f* pState);

IppStatus ippsWTFwd_8s32f(const Ipp8s* pSrc, Ipp32f* pDstLow, Ipp32f* pDstHigh,
    int dstLen, IppsWTFwdState_8s32f* pState);

IppStatus ippsWTFwd_8u32f(const Ipp8u* pSrc, Ipp32f* pDstLow, Ipp32f* pDstHigh,
    int dstLen, IppsWTFwdState_8u32f* pState);

IppStatus ippsWTFwd_16s32f(const Ipp16s* pSrc, Ipp32f* pDstLow, Ipp32f*
    pDstHigh, int dstLen, IppsWTFwdState_16s32f* pState);

IppStatus ippsWTFwd_16u32f(const Ipp16u* pSrc, Ipp32f* pDstLow, Ipp32f*
    pDstHigh, int dstLen, IppsWTFwdState_16u32f* pState);

```

Parameters

<i>pSrc</i>	Pointer to the vector which holds the input signal for decomposition.
<i>pDstLow</i>	Pointer to the vector which holds output coarse “low frequency” components.
<i>pDstHigh</i>	Pointer to the vector which holds output detail “high frequency” components.
<i>dstLen</i>	Number of elements in the vectors <i>pDstHigh</i> and <i>pDstLow</i> .
<i>pState</i>	Pointer to the state structure.

Description

The function `ippsWTFwd` is declared in the `ipps.h` file. This function computes the forward wavelet transform. The function transforms the $(2 * dstLen)$ -length source data block `pSrc` into “low frequency” components `pDstLow` and “high frequency” components `pDstLow`. The transform parameters are specified in the state structure `pState`.

Application Notes

These functions perform the one-level forward discrete multi-scale transform. An equivalent transform diagram is shown in [Figure 7-4](#). The input signal is divided into the “low frequency” and “high frequency” components. The transfer characteristics of filters are defined by the coefficients set at the initialization stage. The functions are designed for the block processing of data; the transform state structure `pState` contains all needed filter delay lines. Besides these main delay lines each function has an additional delay line for each filter. Adjustable extra delay lines help synchronize group delay times of both highpass and lowpass filters. Moreover, in multilevel systems of signal decomposition delays between different decomposition levels may also be synchronized.

Input and output data block lengths. The functions are designed to decompose signal blocks of even length, therefore, these functions have one parameter only, i.e. the length of input components. The length of the input block must be double the size of each component.

Filter group delays synchronization. Some applications may require synchronization of highpass and lowpass filter time responses. A typical example of this synchronization is synchronizing symmetrical filters of different length.

Below follows an example of bi orthogonal set of spline filters of respective length of 6 and 2:

```
static const float decLow[6] =
{
    -6.25000000e-002f,
    6.25000000e-002f,
    5.00000000e-001f,
    5.00000000e-001f,
    6.25000000e-002f,
    -6.25000000e-002f
};

static const float decHigh[2] =
{
    -5.00000000e-001f,
```

```
5.00000000e-001f
```

```
};
```

In this case the lowpass filter gives a delay two samples longer than the highpass filter, which is exactly what the difference between additional initialization function delays should be. The following values must be selected to ensure minimum common signal delay, $offsLow = -1$, $offsHigh = -1 + 2 = 1$. In this case the group times of filter delays are balanced by additional delays. The total delay time is equal to the lowpass filter group delay which has the value of two samples in the decomposition stage in the original signal time frame.



NOTE. Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, forward transform additional delays must be uniformly even for faultless signal reconstruction.

Multilevel decomposition algorithm. The implementation of multilevel decomposition algorithms may require synchronization of signal delays across components of different levels.

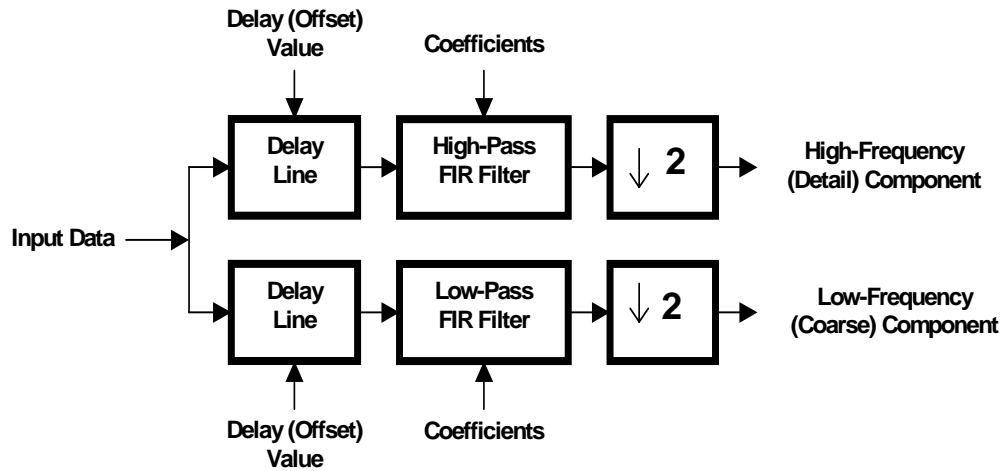
This is illustrated in the example of the three-level decomposition shown in [Figure 7-2](#). Assume that for transformation the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. Since group delay definitely needs to be synchronized, for the last level select additional filter delays $offsLow3 = -1$, $offsHigh3 = 1$. Total delay at the last stage of decomposition for this set of filters is two samples. This value corresponds to the time scale of the input of the last stage of decomposition. In order to ensure an equivalent delay of the “detail” part on the second level, the delay must be increased by $2*2$ samples. Respective values of additional delays for the second level is equal to $offsLow2 = -1$, $offsHigh2 = offsHigh3 + 4 = 5$. A greater value of the “high frequency” component delay needs to be selected for the first level of decomposition, $offsLow1 = -1$, $offsHigh1 = offsHigh2 + 2*4 = 13$.

Total delay for three levels of decomposition is equal to 12 samples.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier <code>pState</code> is incorrect.
<code>ippStsSizeErr</code>	Indicates an error when <code>dstLen</code> or <code>srcLen</code> is less than or equal to 0.

Figure 7-4 One Level Forward Wavelet Transform



WTFwdSetDlyLine, WTFwdGetDlyLine

Sets and gets the delay lines of the forward wavelet transform.

Syntax

```

IppStatus ippsWTFwdSetDlyLine_32f(IppsWTFwdState_32f* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8s32f(IppsWTFwdState_8s32f* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8u32f(IppsWTFwdState_8u32f* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_16s32f(IppsWTFwdState_16s32f* pState, const
    Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
  
```

```
IppStatus ippsWTFwdSetDlyLine_16u32f(IppsWTFwdState_16u32f* pState, const
    Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTFwdGetDlyLine_32f(IppsWTFwdState_32f* pState, Ipp32f* pDlyLow,
    Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTFwdGetDlyLine_8s32f(IppsWTFwdState_8s32f* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTFwdGetDlyLine_8u32f(IppsWTFwdState_8u32f* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTFwdGetDlyLine_16s32f(IppsWTFwdState_16s32f* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTFwdGetDlyLine_16u32f(IppsWTFwdState_16u32f* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
```

Parameters

<i>pState</i>	Pointer to the state structure.
<i>pDlyLow</i>	Pointer to the vector which holds the delay lines for “low frequency” components.
<i>pDlyHigh</i>	Pointer to the vector which holds the delay lines for “high frequency” components.

Description

The functions `ippsWTFwdSetDlyLine` and `ippsWTFwdSetDlyLine` are declared in the `ipps.h` file. These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

ippsWTFwdSetDlyLine. The functions `ippsWTFwdSetDlyLine` sets the delay line values of the forward WT state.

ippsWTFwdGetDlyLine. The functions `ippsWTFwdSetDlyLine` sets the delay line values of the inverse WT state.

Application Notes

These functions are designed to shape the signal prehistory, save and reconstruct delay lines. Delay lines are implemented separately for highpass and lowpass filters, which gives the option of getting independent signal prehistories for each filter.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the initial signal fed into the forward transform functions, i.e., delay line vectors must be made up of a succession of the signal prehistory counts in the same time frame as the initial signal.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter:

$$dlyLowLen = lenLow + offsLow - 1,$$

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the “high frequency” component filter:

$$dlyHighLen = lenHigh + offsHigh - 1,$$

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the “high frequency” component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are specified by the functions [ippsWTFwdInitAlloc](#) and [ippsWTInvInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDlyLow</i> or <i>pDlyHigh</i> is NULL.
<code>ippStsStateMatchErr</code>	Indicates an error when the state identifier <i>pState</i> is incorrect.

WTInv

Computes the inverse wavelet transform.

Syntax

```
IppStatus ippsWTInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp32f* pDst, IppsWTInvState_32f* pState);
IppStatus ippsWTInv_32f8s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp8s* pDst, IppsWTInvState_32f8s* pState);
IppStatus ippsWTInv_32f8u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp8u* pDst, IppsWTInvState_32f8u* pState);
IppStatus ippsWTInv_32f16s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp16s* pDst, IppsWTInvState_32f16s* pState);
IppStatus ippsWTInv_32f16u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp16u* pDst, IppsWTInvState_32f16u* pState);
```

Parameters

<i>pSrcLow</i>	Pointer to the vector which holds input coarse “low frequency” components.
<i>pSrcHigh</i>	Pointer to the vector which holds detail “high frequency” components.
<i>srcLen</i>	Number of elements in the vectors <i>pSrcHigh</i> and <i>pSrcLow</i> .
<i>pDst</i>	Pointer to the vector which holds the output reconstructed signal.
<i>pState</i>	Pointer to the state structure.

Description

The function `ippsWTInv` is declared in the `ipps.h` file. This function computes the inverse wavelet transform. The function transforms the “low frequency” components *pSrcLow* and “high frequency” components *pSrcHigh* into the $(2 * srcLen)$ -length destination data block *pDst*. The transform parameters are specified in the state structure *pState*.

Application Notes

These functions are used for one level of inverse multiscale transformation which results in reconstructing the original signal from the two “low frequency” and “high frequency” components. [Figure 7-5](#) shows an equivalent transform algorithm. Two interpolation filters are

used for signal reconstruction; their coefficients are set at the initialization stage. The inverse transform implementation, similar to forward transform implementation, contains additional delay lines needed to synchronize the group time of filter delays and delays across different levels of data reconstruction.

Input and output data block lengths. These functions are designed to reconstruct the blocks of the even length signal. The signal component length must be the input data. The length of the output block of the reconstructed signal must be double the length of each of the components.

Filter group delay synchronization. In this example consider a biorthogonal set of spline filters of length 2 and 6:

```
static const float recLow[2] =
{
    1.00000000e+000f,
    1.00000000e+000f
};
static const float recHigh[6] =
{
    -1.25000000e-001f,
    -1.25000000e-001f,
    1.00000000e+000f,
    -1.00000000e+000f,
    1.25000000e-001f,
    1.25000000e-001f
};
```

This set of filters corresponds to the set of filters considered in a similar section of the description of the forward transform function [ippsWTFwd](#).

Unlike the case described above, this time the highpass filter generates a delay greater by two samples compared against the low frequency filter. The two sample difference should also exist between initialization function additional delays. The following parameters of additional delays need to be selected in order to ensure the minimum total delay, *offsLow* = 2, *offsHigh* = 0. In this case the total delay is equal to the highpass filter group delay, which at the decomposition stage is equal to two samples in the original signal time frame.

Total delay of one level of decomposition and reconstruction is equal to 4 samples, considering the decomposition stage delay.



NOTE. Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, inverse transform additional delays must be uniformly even and opposite to the evenness of the decomposition delays for faultless signal reconstruction.

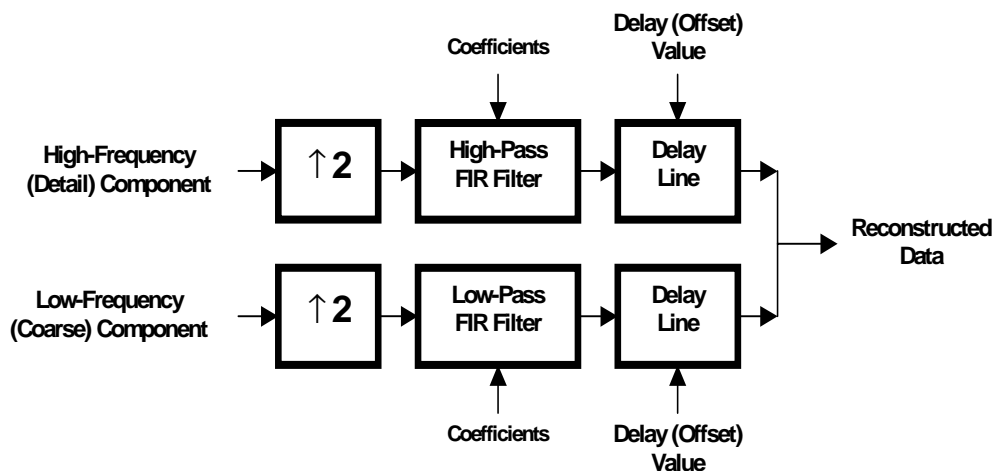
Multilevel reconstruction algorithms. An example of a three-level signal reconstruction algorithm is shown in [Figure 7-3](#). The scheme corresponds to the decomposition scheme described in the section of the description of the forward transform function [ippSWTFwd](#). Therefore, for the inverse transform the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. The lowest level filter delays are set to $offsLow3 = 2$, $offsHigh3 = 0$. The total delay at this stage of reconstruction is equal to two samples. In order to ensure an equivalent delay of the “detail” part in the middle level, the delay must be increased. Respective values of additional delays for the second level are equal to $offsLow2 = 2$, $offsHigh2 = offsHigh3 + 2*2 = 4$. A greater value of high frequency component delay needs to be selected for the last level of reconstruction, $offsLow1 = -1$, $offsHigh1 = offsHigh2 + 2*4 = 12$.

The total delay for three levels of reconstruction is equal to 12 samples. The total delay of the three-level decomposition and reconstruction cycle is equal to 24 samples.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsStateMatchErr</code>	Indicates an error when the state identifier <i>pState</i> is incorrect.
<code>ippStsSizeErr</code>	Indicates an error when <i>dstLen</i> or <i>srcLen</i> is less than or equal to 0.

Figure 7-5 One Level Inverse Wavelet Transform



WTInvSetDlyLine, WTInvGetDlyLine

Sets and gets the delay lines of the inverse wavelet transform.

Syntax

```

IppStatus ippsWTInvSetDlyLine_32f(IppsWTInvState_32f* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8s(IppsWTInvState_32f8s* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8u(IppsWTInvState_32f8u* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f16s(IppsWTInvState_32f16s* pState, const
    Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
  
```

```
IppStatus ippsWTInvSetDlyLine_32f16u(IppsWTInvState_32f16u* pState, const
    Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f(IppsWTInvState_32f* pState, Ipp32f* pDlyLow,
    Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f8s(IppsWTInvState_32f8s* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f8u(IppsWTInvState_32f8u* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f16s(IppsWTInvState_32f16s* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f16u(IppsWTInvState_32f16u* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
```

Parameters

<i>pState</i>	Pointer to the state structure.
<i>pDlyLow</i>	Pointer to the vector which holds delay lines for “low frequency” components.
<i>pDlyHigh</i>	Pointer to the vector which holds delay lines for “high frequency” components.

Description

The functions `ippsWTFwdSetDlyLine` and `ippsWTFwdSetDlyLine` are declared in the `ipps.h` file. These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

ippsWTFwdSetDlyLine. The functions `ippsWTFwdSetDlyLine` sets the delay line values of the forward WT state.

ippsWTInvSetDlyLine. The functions `ippsWTInvSetDlyLine` sets the delay line values of the inverse WT state.

Application Notes

These functions set and read delay lines of inverse multiscale transformation. The functions receive or return filter low and high frequency component delay line vectors. The functions may be used to shape previous history of each of the components. Installation functions and read functions together ensure that delay lines from each filter are saved and reconstructed.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the low and high frequency components at the input of the inverse transform functions. Thus, delay line vectors must be made up of a succession of signal prehistory counts in the same time frame as the input components.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter in terms of the C language (integer division by two is used here for simplicity):

$$dlyLowLen = (lenLow + offsLow - 1) / 2,$$

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the “high frequency” component filter in terms of the C language:

$$dlyHighLen = (lenHigh + offsHigh - 1) / 2,$$

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the “high frequency” component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are also described in the section on the transform initialization, see the functions [ippsWTFwdInitAlloc](#), [ippsWTInvInitAlloc](#) in this chapter.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDlyLow</i> or <i>pDlyHigh</i> is NULL.
<code>ippStsStateMatchErr</code>	Indicates an error when the state identifier <i>pState</i> is incorrect.

Wavelet Transforms Example

The delay line paradigm is well-known interface solution for functions that require some pre-history in the streaming processing. In such application the use of Intel IPP wavelet transform functions is similar to the use of the FIR, IIR, or multi-rate filters. (See also the discussion on the [synchronization](#) of low-pass and high-pass filter delays in this chapter.) But very often the wavelet transforms are used to process entire non-streaming data by extending with borders that are suitable for filter bank type that are used in transforms.

The following code example demonstrates how to implement this approach using Intel IPP functions. It performs forward and inverse wavelet transforms of a short vector containing 12 elements. It uses Daubechies filter bank of the order 2 (that allows the perfect reconstruction) and periodical data extension by wrapping.

It is also may be useful as an illustration of how to fill delay line, if you need non-zero pre-history of signal in streaming applications.

Example 7-11 Using Wavelet Transforms Functions

```
#include <stdio.h>
#include "ipp.h"

//    Filter bank for Daubechies, order 2

static const int fwdFltLenL = 4;
static const int fwdFltLenH = 4;

static const float fwdFltL[4] =
{
    -1.294095225509215e-001f,
    2.241438680418574e-001f,
    8.365163037374690e-001f,
    4.829629131446903e-001f
};

static const float fwdFltH[4] =
{
    -4.829629131446903e-001f,
    8.365163037374690e-001f,
    -2.241438680418574e-001f,
    -1.294095225509215e-001f
};
```

Example 7-11 Using Wavelet Transforms Functions

```
static const int invFltLenL = 4;
static const int invFltLenH = 4;

static const float invFltL[4] =
{
    4.829629131446903e-001f,
    8.365163037374690e-001f,
    2.241438680418574e-001f,
    -1.294095225509215e-001f
};

static const float invFltH[4] =
{
    -1.294095225509215e-001f,
    -2.241438680418574e-001f,
    8.365163037374690e-001f,
    -4.829629131446903e-001f
};

// minimal values
static const int fwdFltOffsL = -1;
static const int fwdFltOffsH = -1;

// minimal values, that corresponds to perfect reconstruction
static const int invFltOffsL = 0;
static const int invFltOffsH = 0;
void main(void)
{
    Ipp32f src[] = {1, -10, 324, 48, -483, 4, 7, -5532, 34, 8889, -57, 54};

    Ipp32f low[7];
    Ipp32f high[7];

    int i;

    printf("original:\n");
    for(i = 0; i < 12; i++) printf("%.0f; ", src[i]);
    printf("\n");

    {
        IppsWTFwdState_32f* state;
```

Example 7-11 Using Wavelet Transforms Functions

```

    ippsWTFwdInitAlloc_32f (&state, fwdFltL, fwdFltLenL, fwdFltOffsL,
                           fwdFltH, fwdFltLenH, fwdFltOffsH);

    // We substitute wrapping extension in "the beginning of stream"
    // Here should be the same pointers for this offsets,
    // but in the general case it may be different

    ippsWTFwdSetDlyLine_32f( state, &src[10], &src[10]);

    // Forward transform

    ippsWTFwd_32f          (src, low, high, 6, state);

    ippsWTFwdFree_32f      (state);

    // print decomposition result

    printf("approx:\n");
    for(i = 0; i < 6; i++) printf("%.4f; ", low[i]);
    printf("\n");

    printf("details:\n");
    for(i = 0; i < 6; i++) printf("%.4f; ", high[i]);
    printf("\n");

    }
    {
        Ipp32f dst[12];

        IppsWTInvState_32f* state;

        ippsWTInvInitAlloc_32f (&state,
                                invFltL, invFltLenL, invFltOffsL,
                                invFltH, invFltLenH, invFltOffsH);

        // For this particular case (non-shifted reconstruction)
        // here is first data itself,
        // that we need to place to delay line
        // [(invFltLenL + invFltOffsL - 1) / 2] elements for l. filtering
        // [(invFltLenH + invFltOffsH - 1) / 2] elements for h. filtering

        ippsWTInvSetDlyLine_32f( state, low, high);
    }

```

Example 7-11 Using Wavelet Transforms Functions

```
// Inverse transform
ippsWTInv_32f          (&low[1], &high[1], 5, dst, state);

// Here are the substitution of the wrapping extension
// at the "end of stream" and calculation of last samples of reconstruction
// We do not use additional buffer and do not copy any data externally,
// just substitute beginning of input data itself to simulate wrapping

ippsWTInv_32f          (low, high, 1, &dst[10], state);

ippsWTInvFree_32f      (state);

// print reconstruction result
printf("reconstruction:\n");
for(i = 0; i < 12; i++) printf("%.0f; ", dst[i]);
printf("\n");
}
}
```

After compiling and running it gives the following console output:

```
original:
1; -10; 324; 48; -483; 4; 7; -5532; 34; 8889; -57; 54;
approx:
19.1612; 58.5288; 87.8536; 487.5375; -5766.9277; 7432.4502;
details:
0.9387; 249.9611; -458.6568; 2739.2146; -3025.5576; -2070.5762;
reconstruction:
1; -10; 324; 48; -483; 4; 7; -5532; 34; 8889; -57; 54;
```

The program prints on console the original data, approximation, and details components after forward transform and perfect reconstruction of original data after inverse transform.

Speech Recognition Functions

8

This chapter describes Intel[®] IPP functions that are designed for use in speech recognition applications.

[Table 8-1](#) gives the full list of functions in this group.

Table 8-1 Intel IPP Speech Recognition Functions

Function Base Name	Operation
Basic Arithmetics Functions	
AddAllRowSum	Calculates the sums of column vectors in a matrix and adds the sums to a vector.
SumColumn	Calculates sums of column vectors in a matrix.
SumRow	Calculates sums of row vectors in a matrix.
SubRow	Subtracts a vector from all matrix rows.
CopyColumn_Indirect	Copies the input matrix with columns redirection.
BlockDMatrixInitAlloc	Initializes the structure that represents a symmetric block diagonal matrix.
BlockDMatrixFree	Deallocates the block diagonal matrix structure.
NthMaxElement	Searches for the N-th maximal element of a vector.
VecMatMul	Multiplies a vector by a matrix.
MatVecMul	Multiplies a matrix by a vector
Feature Processing Functions	
ZeroMean	Subtracts the mean value from the input vector.
CompensateOffset	Removes the DC offset of the input signals.
SignChangeRate	Counts the zero-cross rate for the input signal.
LinearPrediction	Performs linear prediction analysis on the input vector.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
<u>Durbin</u>	Performs Durbin's recursion on an input vector of autocorrelations.
<u>Schur</u>	Calculates reflection coefficients using Schur algorithm.
<u>LPToSpectrum</u>	Calculates smoothed magnitude spectrum.
<u>LPToCepstrum</u>	Calculates cepstrum coefficients from linear prediction coefficients.
<u>CepstrumToLP</u>	Calculates linear prediction coefficients from cepstrum coefficients.
<u>LPToReflection</u>	Calculates the linear prediction reflection coefficients from the linear prediction coefficients.
<u>ReflectionToLP</u>	Calculates the linear prediction coefficients from the linear prediction reflection coefficients.
<u>ReflectionToAR</u>	Converts reflection coefficients to area ratios.
<u>ReflectionToTilt</u>	Calculates tilt for rise/fall/connection parameters.
<u>PitchmarkToF0</u>	Calculates rise and fall amplitude and duration for tilt.
<u>UnitCurve</u>	Calculates tilt for rise and fall coefficients.
<u>LPToLSP</u>	Calculates line spectrum pairs vector from linear prediction coefficients.
<u>LSPToLP</u>	Converts line spectrum pairs vector to linear prediction coefficients.
<u>MelToLinear</u>	Converts Mel-scaled values to linear scale values.
<u>LinearToMel</u>	Converts linear-scale values to Mel-scale values.
<u>CopyWithPadding</u>	Copies the input signal to the output with zero-padding.
<u>MelFBankGetSize</u>	Gets the size of the Mel-frequency filter bank structure.
<u>MelFBankInit</u>	Initializes the structure for performing the Mel-frequency filter bank analysis.
<u>MelFBankInitAlloc</u>	Initializes the structure and allocates memory for performing the Mel-frequency filter bank analysis.
<u>MelLinFBankInitAlloc</u>	Initializes the structure for performing a combined linear and Mel-frequency filter bank analysis.
<u>EmptyFBankInitAlloc</u>	Initializes an empty filter bank structure.
<u>FBankFree</u>	Destroys the structure for the filter bank analysis.
<u>FBankGetCenters</u>	Retrieves the center frequencies of the triangular filter banks.
<u>FBankSetCenters</u>	Sets the center frequencies of the triangular filter banks.
<u>FBankGetCoeffs</u>	Retrieves the filter bank weight coefficients.
<u>FBankSetCoeffs</u>	Sets the filter bank weight coefficients.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
<u>EvalFBank</u>	Performs the filter bank analysis.
<u>DCTLifterGetSize_MulC0</u>	Gets the size of the DCT structure.
<u>DCTLifterInit_MulC0</u>	Initializes the structure to perform DCT and lift the DCT coefficients.
<u>DCTLifterInitAlloc</u>	Initializes the structure and allocates memory to perform DCT and lift the DCT coefficients.
<u>DCTLifterFree</u>	Destroys the structure used for the DCT and lifting.
<u>DCTLifter</u>	Performs the DCT and lifts the DCT coefficients.
<u>NormEnergy</u>	Normalizes a vector of energy values.
<u>SumMeanVar</u>	Calculates both the sum of a the vector and its square sum.
<u>NewVar</u>	Calculates the variances given the sum and square sum accumulators.
<u>RecSqrt</u>	Calculates square roots of a vector and their reciprocals.
<u>AccCovarianceMatrix</u>	Accumulates covariance matrix.
Derivative Functions	
<u>CopyColumn</u>	Copies the input sequence into the output sequence.
<u>EvalDelta</u>	Calculates the derivatives of feature vectors.
<u>Delta</u>	Copies the base features and calculates the derivatives of feature vectors.
<u>DeltaDelta</u>	Copies the base features and calculates their first and second derivatives.
Pitch Super Resolution Functions	
<u>CrossCorrCoeffDecim</u>	Calculates vector of cross correlation coefficients with decimation.
<u>CrossCorrCoeff</u>	Calculates the cross correlation coefficient.
<u>CrossCorrCoeffInterpolation</u>	Calculates interpolated cross correlation coefficient.
Model Evaluation Functions	
<u>AddNRows</u>	Adds N vectors from a vector array.
<u>ScaleLM</u>	Scales vector elements with thresholding.
<u>LogAdd</u>	Adds two vectors in the logarithmic representation.
<u>LogSub</u>	Subtracts a vector from another vector, in the logarithmic representation.
<u>LogSum</u>	Sums vector elements in the logarithmic representation.
<u>MahDistSingle</u>	Calculates the Mahalanobis distance for a single observation vector.
<u>MahDist</u>	Calculates the Mahalanobis distances for multiple observation vectors.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
<u>MahDistMultiMix</u>	Calculates the Mahalanobis distances for multiple means and variances.
<u>LogGaussSingle</u>	Calculates the observation probability for a single Gaussian with an observation vector.
<u>LogGauss</u>	Calculates the observation probability for a single Gaussian with multiple observation vectors.
<u>LogGaussMultiMix</u>	Calculates the observation probability for multiple Gaussian mixture components.
<u>LogGaussMax</u>	Calculates the likelihood probability given multiple observations and a Gaussian mixture component, using the maximum operation.
<u>LogGaussMaxMultiMix</u>	Calculate the likelihood probability for multiple Gaussian mixture components, using the maximum operation.
<u>LogGaussAdd</u>	Calculates the likelihood probability for multiple observation vectors.
<u>LogGaussAddMultiMix</u>	Calculates the likelihood probability for multiple Gaussian mixture components.
<u>LogGaussMixture</u>	Calculates the likelihood probability for the Gaussian mixture.
<u>LogGaussMixtureSelect</u>	Calculates the likelihood probability for the Gaussian mixture using Gaussian selection.
<u>BuildSignTable</u>	Fills sign table for Gaussian mixture calculation.
<u>FillShortlist_Row</u>	Fills row-wise shortlist table for Gaussian selection.
<u>FillShortlist_Column</u>	Fills column-wise shortlist table for Gaussian selection.
<u>DTW</u>	Computes the distance between observation and reference vector sequences using Dynamic Time Warping algorithm.
Model Estimation Functions	
<u>MeanColumn</u>	Computes the mean values for the column elements.
<u>VarColumn</u>	Calculates the variances for the column elements.
<u>MeanVarColumn</u>	Calculates the means and variances for the column elements of a matrix.
<u>WeightedMeanColumn</u>	Computes the weighted mean values for the column elements.
<u>WeightedVarColumn</u>	Computes the weighted variance values for the column elements.
<u>WeightedMeanVarColumn</u>	Computes weighted mean and variance values for the column elements.
<u>NormalizeColumn</u>	Normalizes the matrix columns given the column means and variances.
<u>NormalizeInRange</u>	Normalizes and scales input vector elements.
<u>MeanVarAcc</u>	Accumulates the estimates for the mean and variance re-estimation.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
<u>GaussianDist</u>	Calculates the distance between two Gaussians.
<u>GaussianSplit</u>	Splits a single Gaussian component into two with the same variance.
<u>GaussianMerge</u>	Merges two Gaussian probability distribution functions.
<u>Entropy</u>	Calculates entropy of the input vector.
<u>SinC</u>	Calculates sine divided by its argument.
<u>ExpNegSqr</u>	Calculates exponential of the squared argument taken with the inverted sign.
<u>BhatDist</u>	Calculates the Bhattacharia distance between two Gaussians.
<u>UpdateMean</u>	Updates the mean vector in the EM training algorithm.
<u>UpdateVar</u>	Updates the variance vector in the EM training algorithm.
<u>UpdateWeight</u>	Updates the weight values of Gaussian mixtures in the EM training algorithm.
<u>UpdateGConst</u>	Updates the fixed constant in the Gaussian output probability density function.
<u>OutProbPreCalc</u>	Pre-calculates the part of Gaussian mixture output probability that is irrelevant to observation vectors.
<u>DcsClustLAccumulate</u>	Updates the accumulators for calculating the state-cluster likelihood in the decision-tree clustering algorithm.
<u>DcsClustLCompute</u>	Calculates the likelihood of an HMM state cluster in the decision-tree state-clustering algorithm.
Model Adaptation Functions	
<u>AddMulColumn</u>	Adds a weighted matrix column to the other column.
<u>AddMulRow</u>	Adds a weighted vector to the other vector.
<u>QRTransColumn</u>	Performs the QR transformation.
<u>DotProdColumn</u>	Calculates the dot product of two matrix columns.
<u>MulColumn</u>	Multiplies a matrix column by a value.
<u>SumColumnAbs</u>	Calculates the absolute sum of matrix column elements.
<u>SumColumnSqr</u>	Calculates the square sums of weighted matrix column elements.
<u>SumRowAbs</u>	Calculates the absolute sum of the vector elements.
<u>SumRowSqr</u>	Calculates the square sum of weighted vector elements.
<u>SVD, SVDSort</u>	Performs Single Value Decomposition on a matrix.
<u>WeightedSum</u>	Calculates the weighted sums of two input vector elements.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
Vector Quantization Functions	
<u>FormVector</u>	Constructs an output vector of multiple streams from codebook entries.
<u>CdbkGetSize</u>	Calculates the size in bytes of the codebook.
<u>CdbkInit</u>	Initializes the structure that contains the codebook.
<u>CdbkInitAlloc</u>	Initializes the codebook structure.
<u>CdbkFree</u>	Destroys the codebook structure.
<u>GetCdbkSize</u>	Retrieves the number of codevectors in the codebook.
<u>GetCodebook</u>	Retrieves the codevectors from the codebook.
<u>VQ</u>	Quantizes the input vectors given a codebook.
<u>SplitVQ</u>	Quantizes a multiple-stream vector given the codebooks.
<u>VQSingle_Sort,</u> <u>VQSingle_Thresh</u>	Quantizes the input vector given a codebook and gets several closest clusters.
<u>FormVectorVQ</u>	Constructs multiple-stream vectors from codebooks, given indexes.
Polyphase Resampling Functions	
<u>ResamplePolyphaseInitAlloc</u>	Initializes the structure for polyphase data resampling.
<u>ResamplePolyphaseFree</u>	Free structure for polyphase data resampling.
<u>ResamplePolyphase</u>	Resamples input data using polyphase filters.
Advanced Aurora Functions	
<u>SmoothedPowerSpectrumAurora</u>	Calculates smoothed magnitude of the FFT output.
<u>NoiseSpectrumUpdate_Aurora</u>	Updates the noise spectrum.
<u>WienerFilterDesign_Aurora</u>	Calculates an improved transfer function of the adaptive Wiener filter.
<u>MelFBankInitAlloc_Aurora</u>	Initializes the structure for performing the Mel-frequency filter bank analysis.
<u>TabsCalculation_Aurora</u>	Calculates filter coefficients for residual filter.
<u>ResidualFilter_Aurora</u>	Calculates a denoised waveform signal.
<u>WaveProcessing_Aurora</u>	Processes waveform data after noise reduction.
<u>LowHighFilter_Aurora</u>	Calculates low band and high band filters.
<u>HighBandCoding_Aurora</u>	Codes and decodes the high frequency band energy values.
<u>BlindEqualization_Aurora</u>	Equalizes the cepstral coefficients.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
<u>DeltaDelta_Aurora</u>	Calculates the first and second derivatives according to ETSI ES 202 050 standard.
<u>VADGetBufSize_Aurora</u>	Queries the memory size for VAD decision.
<u>VADInit_Aurora</u>	Gets the VAD structure size.
<u>VADDecision_Aurora</u>	Takes the VAD decision.
<u>VADFlush_Aurora</u>	Takes VAD decision for zero input frame.
Ephraim-Malah Noise Suppressor Functions	
<u>FilterUpdateEMNS</u>	Calculates the noise suppression filter coefficients.
<u>FilterUpdateWiener</u>	Calculates the Wiener filter coefficients.
<u>GetSizeMCRA</u>	Calculates the size in bytes required for the state structure.
<u>InitMCRA</u>	Initializes the <code>IppMCRASState</code> state structure.
<u>AltInitMCRA</u>	Allocates memory and initializes the <code>IppMCRASState</code> state structure.
<u>UpdateNoisePSDMCRA</u>	Re-estimates the noise power spectrum.
Acoustic Echo Canceller Functions	
<u>FilterAECNLMS</u>	Computes the frequency-domain adaptive filter output.
<u>CoefUpdateAECNLMS</u>	Updates the adaptive filter coefficients.
<u>StepSizeUpdateAECNLMS</u>	Computes the adaptive step size.
<u>ControllerGetSizeAEC</u>	Returns the size of the AEC controller state structure.
<u>ControllerInitAEC</u>	Initializes the AEC controller state structure.
<u>ControllerUpdateAEC</u>	Implements the energy-based AEC controller.
Voice Activity Detector Functions	
<u>FindPeaks</u>	Identifies peaks in the input vector.
<u>PeriodicityLSPE</u>	Computes the periodicity of the input speech frame.
<u>Periodicity</u>	Computes the periodicity of the input block.



NOTE. In the below descriptions of function arguments, when an argument refers to a vector or an array, the expression in square brackets given after the explanation of the argument specifies the dimension of that vector or array.

In this chapter, unlike other chapters of this manual, arguments that represent step values are measured in elements of the corresponding array, unless explicitly stated to the contrary.

The use of the Intel IPP speech recognition functions is demonstrated in Intel IPP Samples. See *Intel IPP Speech Recognition Samples* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Basic Arithmetics

The functions included in this section perform generic arithmetic operations on vectors and matrices.

AddAllRowSum

Calculates the sums of column vectors in a matrix and adds the sums to a vector.

Syntax

```
IppStatus ippsAddAllRowSum_32f_D2(const Ipp32f* pSrc, int step,
    int height, Ipp32f* pSrcDst, int width);
IppStatus ippsAddAllRowSum_32f_D2L(const Ipp32f** mSrc, int height,
    Ipp32f* pSrcDst, int width);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height</i> * <i>step</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].

<i>step</i>	Row step in <i>pSrc</i> .
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>pSrcDst</i>	Pointer to the output vector [<i>width</i>].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> , and also the length of the output vector <i>pSrcDst</i> .

Description

The function `ippsAddAllRowSum` is declared in the `ippsr.h` file. This function calculates sums of column vectors in the input matrix, and adds these sums to the output vector. The operations are as follows:

For functions with the D2 suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} pSrc[i \cdot step + j], \quad 0 \leq j < width.$$

For functions with the D2L suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} mSrc[i][j], \quad 0 \leq j < width.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

SumColumn

Calculates sums of column vectors in a matrix.

Syntax

```
IppStatus ippsSumColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int step, int height,
    Ipp32s* pDst, int width, int scaleFactor);
```



```

IppStatus ippsSumColumn_16s32f_D2(const Ipp16s* pSrc, int step, int height,
    Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2(const Ipp32f* pSrc, int step, int height,
    Ipp32f* pDst, int width);
IppStatus ippsSumColumn_64f_D2(const Ipp64f* pSrc, int step, int height,
    Ipp64f* pDst, int width);
IppStatus ippsSumColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height,
    Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumColumn_16s32f_D2L(const Ipp16s** mSrc, int height,
    Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f* pDst,
    int width);
IppStatus ippsSumColumn_64f_D2L(const Ipp64f** mSrc, int height, Ipp64f* pDst,
    int width);

```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height*step</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>pDst</i>	Pointer to the output vector [<i>width</i>].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> and the length of the output vector <i>pDst</i> .
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsSumColumn` is declared in the `ippsr.h` file. This function calculates sums of column vectors in the input matrix, and stores these sums in the output vector. The operations are as follows:

For functions with the D2 suffix,

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i \cdot step + j], \quad 0 \leq j < width.$$

For functions with the D2L suffix,

$$pDst[j] = \sum_{i=0}^{height-1} mSrc[i][j], \quad 0 \leq j < width.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> , <code>mSrc</code> , or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>height</code> or <code>width</code> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <code>step</code> is less than <code>width</code> .

SumRow

Calculates sums of row vectors in a matrix.

Syntax

```
IppStatus ippsSumRow_16s32s_D2Sfs(const Ipp16s* pSrc, int width, int step,
    Ipp32s* pDst, int height, int scaleFactor);
IppStatus ippsSumRow_16s32f_D2(const Ipp16s* pSrc, int width, int step, Ipp32f*
    pDst, int height);
IppStatus ippsSumRow_32f_D2(const Ipp32f* pSrc, int width, int step,
    Ipp32f* pDst, int height);
IppStatus ippsSumRow_64f_D2(const Ipp64f* pSrc, int width, int step,
    Ipp64f* pDst, int height);
IppStatus ippsSumRow_16s32s_D2LSfs(const Ipp16s** mSrc, int width, Ipp32s*
    pDst, int height, int scaleFactor);
IppStatus ippsSumRow_16s32f_D2L(const Ipp16s** mSrc, int width, Ipp32f* pDst,
    int height);
IppStatus ippsSumRow_32f_D2L(const Ipp32f** mSrc, int width, Ipp32f* pDst,
    int height);
IppStatus ippsSumRow_64f_D2L(const Ipp64f** mSrc, int width, Ipp64f* pDst,
    int height);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height</i> * <i>step</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> , and also the length of the output vector <i>pDst</i> .
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the output vector [<i>height</i>].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> .
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsSumRow` is declared in the `ippsr.h` file. This function calculates sums of row vectors in the input matrix *mSrc* and stores these sums in the output vector *pDst*. The operations are as follows:

For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[i \cdot step + j], \quad 0 \leq i < height.$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} mSrc[i][j], \quad 0 \leq i < height.$$

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , or <i>pDst</i> pointer is NULL.
<code>ippsStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippsStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

SubRow

Subtracts a vector from all matrix rows.

Syntax

```
IppStatus ippsSubRow_16s_D2(const Ipp16s* pSrc, int width, Ipp16s* pSrcDst, int
    dstStep, int height);
IppStatus ippsSubRow_32f_D2(const Ipp32f* pSrc, int width, Ipp32f* pSrcDst,
    int dstStep, int height);
IppStatus ippsSubRow_16s_D2L(const Ipp16s* pSrc, Ipp16s** mSrcDst, int width,
    int height);
IppStatus ippsSubRow_32f_D2L(const Ipp32f* pSrc, Ipp32f** mSrcDst, int width,
    int height);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>width</i>].
<i>pSrcDst</i>	Pointer to the source and destination vector [<i>height</i> * <i>dstStep</i>].
<i>mSrcDst</i>	Pointer to the source and destination matrix [<i>height</i>][<i>width</i>].
<i>width</i>	Number of columns in the matrix <i>mSrcDst</i> .
<i>dstStep</i>	Row step in the vector <i>pSrcDst</i> .
<i>height</i>	Number of rows in the matrix <i>mSrcDst</i> .

Description

The function `ippsSubRow` is declared in the `ippsr.h` file. This function subtracts the input vector *pSrc* from all matrix rows. The operations are as follows:

For functions with the D2 suffix,

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] - pSrc[j],$$

$$0 \leq i < height, 0 \leq j < width.$$

For functions with the D2L suffix,

$$mSrcDst[i][j] = mSrcDst[i][j] - pSrc[j],$$

$0 \leq i < height, 0 \leq j < width$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pSrcDst</i> , or <i>mSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>dstStep</i> is less than <i>width</i> .

CopyColumn_Indirect

Copies the input matrix with columns redirection.

Syntax

```

IppStatus ippCopyColumn_Indirect_16s_D2(const Ipp16s* pSrc, int srcLen,
    int srcStep, Ipp16s* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
    height);

IppStatus ippCopyColumn_Indirect_32f_D2(const Ipp32f* pSrc, int srcLen,
    int srcStep, Ipp32f* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
    height);

IppStatus ippCopyColumn_Indirect_64f_D2 (const Ipp64f* pSrc, int srcLen, int
    srcStep, Ipp64f* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
    height);

IppStatus ippCopyColumn_Indirect_16s_D2L(const Ipp16s** mSrc, int srcLen,
    Ipp16s** mDst, const Ipp32s* pIndx, int dstLen, int height);

IppStatus ippCopyColumn_Indirect_32f_D2L(const Ipp32f** mSrc, int srcLen,
    Ipp32f** mDst, const Ipp32s* pIndx, int dstLen, int height);

IppStatus ippCopyColumn_Indirect_64f_D2L (const Ipp64f** mSrc, int srcLen,
    Ipp64f** mDst, const Ipp32s* pIndx, int dstLen, int height);

```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height</i> * <i>srcStep</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>srcLen</i>].
<i>srcLen</i>	Number of columns in the input matrix <i>mSrc</i> .

<i>srcStep</i>	Row step in the vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the output vector [<i>height</i> * <i>dstStep</i>].
<i>mDst</i>	Pointer to the output matrix [<i>height</i>][<i>dstLen</i>].
<i>pIndx</i>	Pointer to the redirection vector [<i>dstLen</i>].
<i>dstLen</i>	Number of columns in the output matrix <i>mDst</i> .
<i>dstStep</i>	Row step in the vector <i>pDst</i> .
<i>height</i>	Number of rows in both the input and output matrices.

Description

The function `ippsCopyColumn_Indirect` is declared in the `ippsr.h` file. This function copies the input matrix *mSrc* to the output matrix *mDst* with the columns redirected by *pIndx*. The operations are as follows:

For functions with the D2 suffix,

$$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcStep + pIndx[j]], \quad 0 \leq i < height, \quad 0 \leq j < dstLen.$$

For functions with the D2L suffix,

$$mDst[i][j] = mSrc[i][pIndx[j]], \quad 0 \leq i < height, \quad 0 \leq j < dstLen.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , <i>pDst</i> , <i>mDst</i> , or <i>pIndx</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>srcLen</i> , or <i>dstLen</i> is less than or equal to 0, or $pIndx[j] \geq srcLen$ or $pIndx[j] < 0$ for $0 \leq j < dstLen$.
<code>ippStsStrideErr</code>	Indicates an error when <i>srcStep</i> is less than <i>srcLen</i> or <i>dstStep</i> is less than <i>dstLen</i> .

BlockDMatrixInitAlloc

Initializes the structure that represents a symmetric block diagonal matrix.

Syntax

```
IppStatus ippsBlockDMatrixInitAlloc_16s(IppsBlockDMatrix_16s** pMatrix,
    const Ipp16s** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_32f(IppsBlockDMatrix_32f** pMatrix,
    const Ipp32f** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_64f(IppsBlockDMatrix_64f** pMatrix,
    const Ipp64f** mSrc, const int* bSize, int nBlocks);
```

Parameters

<i>pMatrix</i>	Pointer to the block diagonal matrix to be created.
<i>mSrc</i>	Pointer to the vector of pointers to matrix rows.
<i>bSize</i>	Pointer to vector of block sizes [<i>nBlocks</i>].
<i>nBlocks</i>	Number of blocks in the matrix.

Description

The function `ippsBlockDMatrixInitAlloc` is declared in the `ippsr.h` file. This function creates the structure that contains a symmetric block diagonal matrix. Matrix elements outside the blocks are assumed to be zero. The block diagonal matrix A is defined as follows:

$$A[K_l + i][K_l + j] = A[K_l + j][K_l + i] = mSrc[K_l + i][j],$$

for $i, j = 0 \dots bSize[l] - 1$ and $l = 0 \dots nBlocks - 1$,

where

$$K_l = \sum_{k < l} bSize[k]$$

and $mSrc[K_l + i]$ points to the non-zero part of the matrix A row.

The size of the matrix is equal to $K_{nBlocks}$ by $K_{nBlocks}$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pMatrix</i> or <i>mSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>bSize</i> or <i>nBlocks</i> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.

BlockDMatrixFree

Deallocates the block diagonal matrix structure.

Syntax

```
IppStatus ippBlockDMatrixFree_16s(IppsBlockDMatrix_16s* pMatrix);  
IppStatus ippBlockDMatrixFree_32f(IppsBlockDMatrix_32f* pMatrix);  
IppStatus ippBlockDMatrixFree_64f(IppsBlockDMatrix_64f* pMatrix);
```

Parameters

pMatrix Pointer to the block diagonal matrix.

Description

The function `ippBlockDMatrixFree` is declared in the `ippsr.h` file. This function destroys the block diagonal matrix structure, and frees all memory associated with it.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pMatrix</i> pointer is NULL.

NthMaxElement

Searches for the N -th maximal element of a vector.

Syntax

```
IppStatus ippsNthMaxElement_32s(const Ipp32s* pSrc, int len, int N,
                                Ipp32s* pRes);
IppStatus ippsNthMaxElement_32f(const Ipp32f* pSrc, int len, int N,
                                Ipp32f* pRes);
IppStatus ippsNthMaxElement_64f(const Ipp64f* pSrc, int len, int N,
                                Ipp64f* pRes);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len</i>].
<i>len</i>	Number of elements in the input vector <i>pSrc</i> .
<i>N</i>	Rank of element to find.
<i>pRes</i>	Pointer to the value of N -th maximal element.

Description

The function `ippsNthMaxElement` is declared in the `ippsr.h` file. This function finds the N -th maximal element of the input vector *pSrc*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pRes</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates an error when <i>N</i> is less than 0, or greater than or equal to <i>len</i> .

VecMatMul

Multiplies a vector by a matrix.

Syntax

```
IppStatus ippsVecMatMul_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pMatr, int
    step, int height, Ipp16s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_16s_D2LSfs(const Ipp16s* pSrc, const Ipp16s** mMatr,
    int height, Ipp16s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_32f_D2(const Ipp32f* pSrc, const Ipp32f* pMatr, int
    step, int height, Ipp32f* pDst, int width);
IppStatus ippsVecMatMul_32f_D2L(const Ipp32f* pSrc, const Ipp32f** mMatr, int
    height, Ipp32f* pDst, int width);
IppStatus ippsVecMatMul_16s32s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pMatr,
    int step, int height, Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_16s32s_D2LSfs(const Ipp16s* pSrc, const Ipp16s* pMatr,
    int step, int height, Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_32s_D2Sfs(const Ipp32s* pSrc, const Ipp32s* pMatr, int
    step, int height, Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_32s_D2LSfs(const Ipp32s* pSrc, const Ipp32s* pMatr, int
    step, int height, Ipp32s* pDst, int width, int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height</i>].
<i>pMatr</i>	Pointer to the input matrix vector [<i>height</i> * <i>step</i>].
<i>mMatr</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in <i>pMatr</i> vector (in <i>pMatr</i> elements).
<i>width</i>	Length of the result vector and input matrix rows.
<i>height</i>	Number of rows in the input matrix.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsVecMatMul` is declared in the `ippsr.h` file. This function multiplies the input row vector by the given matrix as:

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot pMatr[i \cdot step + j]$$

for the function flavors with `D2` suffix,

and

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot mMatr[i][j]$$

for the function flavors with `D2L` suffix,

where $0 \leq j < width$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> , <code>pMatr</code> , <code>mMatr</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>width</code> or <code>height</code> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <code>step</code> is less than <code>width</code> .

MatVecMul

Multiplies a matrix by a vector.

Syntax

```
IppStatus ippsMatVecMul_16s_D2Sfs(const Ipp16s* pMatr, int step, const Ipp16s*
    pSrc, int width, Ipp16s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_16s_D2LSfs(const Ipp16s** mMatr, const Ipp16s* pSrc,
    int width, Ipp16s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_32f_D2(const Ipp32f* pMatr, int step, const Ipp32f*
    pSrc, int width, Ipp32f* pDst, int height);
```

```

IppStatus ippsMatVecMul_32f_D2L(const Ipp32f** mMatr, const Ipp32f* pSrc, int
    width, Ipp32f* pDst, int height);
IppStatus ippsMatVecMul_16s32s_D2Sfs(const Ipp16s* pMatr, int step, const
    Ipp16s* pSrc, int width, Ipp32s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_16s32s_D2LSfs(const Ipp16s** mMatr, const Ipp16s* pSrc,
    int width, Ipp32s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_32s_D2Sfs(const Ipp32s* pMatr, int step, const Ipp32s*
    pSrc, int width, Ipp32s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_32s_D2LSfs(const Ipp32s** mMatr, const Ipp32s* pSrc,
    int width, Ipp32s* pDst, int height, int scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>width</i>].
<i>pMatr</i>	Pointer to the input matrix vector [<i>height</i> * <i>step</i>].
<i>mMatr</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in <i>pMatr</i> vector (in <i>pMatr</i> elements).
<i>width</i>	Length of the input vector and input matrix rows.
<i>height</i>	Number of rows in the input matrix and the length of the result vector.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsVecMatMul` is declared in the `ippsr.h` file. This function multiplies the the given matrix by the input column vector as:

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot pMatr[i \cdot step + j]$$

for the function flavors with D2 suffix,

and

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot mMatr[i][j]$$

for the function flavors with D2L suffix,

where $0 \leq i < height$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pMatr</i> , <i>mMatr</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

Feature Processing

This section describes functions that pre-process raw speech signals. Feature extraction is the first step in the recognition process. Speech features are a compressed form of the original speech signals. By extracting features, the problem dimension is reduced. To some extent, feature extraction normalizes speaker variations and environmental distortions.

This section also includes functions needed to support both silence/speech detection and also some well-known analysis techniques on speech signals.

This functions are used in the Intel IPP Speech Recognition Samples. See *Aurora Encoder-Decoder Sample* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

ZeroMean

Subtracts the mean value from the input vector.

Syntax

```
IppStatus ippsZeroMean_16s(Ipp16s* pSrcDst, int len);
```

Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector [<i>len</i>].
<i>len</i>	The number of elements in the vector.

Description

The function `ippsZeroMean` is declared in the `ippsr.h` file. This function calculates the mean value of the `pSrcDst` vector and subtracts it from the vector `pSrcDst`. The resulting values are saturated if they exceed the range `[-32768..32767]`.

The operations are as follows:

$$pSrcDst[i] = \max(-32768, \min(32767, pSrcDst[i] - \frac{1}{len} \sum_{j=0}^{len-1} pSrcDst[j]))$$

for $0 \leq i < len$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

CompensateOffset

Removes the DC offset of the input signals.

Syntax

```
IppStatus ippsCompensateOffset_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s* pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f* pSrcDst0, Ipp32f dst0, Ipp32f val);
IppStatus ippsCompensateOffset_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
    pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f_I(Ipp32f* pSrcDst, int len, Ipp32f*
    pSrcDst0, Ipp32f dst0, Ipp32f val);
IppStatus ippsCompensateOffsetQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
IppStatus ippsCompensateOffsetQ15_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
    pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
```

Parameters

<i>pSrc</i>	Pointer to the source vector [<i>len</i>].
<i>pDst</i>	Pointer to the destination vector [<i>len</i>].
<i>pSrcDst</i>	Pointer to the source and destination vector [<i>len</i>] for in-place operations.
<i>pSrcDst0</i>	Pointer to the previous source element.
<i>len</i>	Number of elements in the vector.
<i>dst0</i>	Previous destination element.
<i>val</i>	Constant for offset compensation.
<i>valQ15</i>	Real-valued Q15 format constant for offset compensation ($0.0_{Q31} \leq valQ15 < 1.0_{Q31}$).
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsCompensateOffset` is declared in the `ippsr.h` file. This function removes the offset of the input signals. The destination vector is calculated as follows.

For the `ippsCompensateOffset` function:

$$pDst[0] = pSrc[0] - pSrcDst[0] + val \cdot dst0$$

$$pDst[i] = pSrc[i] - pSrc[i-1] + val \cdot pDst[i-1], 1 \leq i \leq len-1$$

$$pSrcDst0[0] = pSrc[len-1],$$

and for `ippsCompensateOffset_I` function:

$$y = pSrcDst[0] - pSrcDst0[0] + val \cdot dst0, x = pSrcDst[0], pSrcDst[0] = y,$$

$$y = pSrcDst[i] - pSrcDst[i-1] + val \cdot x, x = pSrcDst[i], pSrcDst[i] = y,$$

$$1 \leq i \leq len-1,$$

$$pSrcDst0[0] = x.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> , <code>pDst</code> , <code>pSrcDst</code> , or <code>pSrcDst0</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0, or when <code>valQ15</code> is out of range.

SignChangeRate

Counts the zero-cross rate for the input signal.

Syntax

```
IppStatus ippSignChangeRate_16s(const Ipp16s* pSrc, int len, Ipp32s* pRes);
IppStatus ippSignChangeRate_32f(const Ipp32f* pSrc, int len, Ipp32f* pRes);
IppStatus ippSignChangeRate_Count0_16s(const Ipp16s* pSrc, int len, Ipp32s* pRes);
IppStatus ippSignChangeRate_Count0_32f(const Ipp32f* pSrc, int len, Ipp32f* pRes);
IppStatus ippSignChangeRateXor_32f(const Ipp32f* pSrc, int len, Ipp32f* pRes);
```

Parameters

<code>pSrc</code>	Pointer to the input signal [<code>len</code>].
<code>len</code>	Number of elements in the input signal <code>pSrc</code> .
<code>pRes</code>	Pointer to the result variable.

Description

The function `ippSignChangeRate` is declared in the `ippsr.h` file. This function counts the number of sign changes in the input signal and can be used to detect speech in continuous speech input.

The operations are as follows:

For the `ippSignChangeRate` function,

$$pRes[0] = \sum_{i=1}^{len-1} \begin{cases} 1, & \text{if } pSrc[i] \cdot pSrc[i-1] < 0 \\ 0, & \text{otherwise} \end{cases}$$

For the `ippsSignChangeRateCount0` function,

$$pRes[0] = \frac{1}{2} \sum_{i=1}^{len-1} |sign(pSrc[i]) - sign(pSrc[i-1])|, \text{ where } sign(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

For the `ippsSignChangeRateXor` function,

$$pRes[0] = \sum_{i=0}^{len-2} sign(pSrc[i]) \oplus sign(pSrc[i+1]), \text{ where } sign(x) = \begin{cases} 0, & x > 0, x = +0 \\ 1, & x < 0, x = -0 \end{cases}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pRes</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

LinearPrediction

*Performs linear prediction analysis
on the input vector.*

Syntax

```

IppStatus ippsLinearPrediction_Auto_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
    Ipp16s* pDst, int lenDst, int scaleFactor);

IppStatus ippsLinearPrediction_Auto_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
    pDst, int lenDst);

IppStatus ippsLinearPredictionNeg_Auto_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
    Ipp16s* pDst, int lenDst, int scaleFactor);

IppStatus ippsLinearPredictionNeg_Auto_32f(const Ipp32f* pSrc, int lenSrc,
    Ipp32f* pDst, int lenDst);

IppStatus ippsLinearPrediction_Cov_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
    Ipp16s* pDst, int lenDst, int scaleFactor);

IppStatus ippsLinearPrediction_Cov_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
    pDst, int lenDst);

```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>lenSrc</i>].
<i>lenSrc</i>	Length of the input vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the output LPC coefficients vector [<i>lenDst</i>].
<i>lenDst</i>	Length of the output vector <i>pDst</i> .
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsLinearPrediction` is declared in the `ippsr.h` file. This function performs linear prediction analysis on the input signal.

For functions with the `Auto` suffix, the LPC coefficients are calculated by solving the following equations that represent the autocorrelation approach:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = r[k], k = 1, \dots, lenDst,$$

$$\text{where } r[k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k].$$

For functions with the `Neg_Auto` suffix, the LPC coefficients are calculated by solving the similar equations with inverted sign of the right-hand side:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = -r[k], k = 1 \dots lenDst,$$

For functions with the `Cov` suffix, the LPC coefficients are calculated by solving the following equations that represent the covariance approach:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot c[i][k] = c[0][k], k = 1, \dots, lenDst,$$

$$\text{where } c[i][k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k-i] .$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>lenSrc</code> or <code>lenDst</code> is less than or equal to 0, or <code>lenDst</code> is greater or equal than <code>lenSrc</code> .
<code>ippStsNoOperation</code>	Indicates no solution to the LPC problem.

Durbin

*Performs Durbin's recursion
on an input vector of autocorrelations.*

Syntax

```
IppStatus ippsDurbin_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp32f*
    pErr, int scaleFactor);
IppStatus ippsDurbin_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f* pErr);
```

Parameters

<code>pSrc</code>	Pointer to the input vector [<code>len+1</code>].
<code>pDst</code>	Pointer to the output LPC coefficients vector [<code>len</code>].
<code>len</code>	Length of the output vector.
<code>pErr</code>	Pointer to the result prediction error.
<code>scaleFactor</code>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsDurbin` is declared in the `ippsr.h` file. This function performs the Durbin's recursion on the input autocorrelation vector and calculates the linear prediction coefficients and the prediction error as follows:

1) Initialization:

$$m = \text{len}, R_j = pSrc[j], j = 0, \dots, m, E^0 = R_0$$

2) Iterations for $i = 1, \dots, m$:

$$k_i = \left(R_i - \sum_{j=1}^{i-1} Y_j^{i-1} \cdot R_{i-j} \right) / E^{i-1}, E^i = (1 - k_i^2) \cdot E^{i-1},$$

$$Y_i^i = k_i, Y_j^i = Y_j^{i-1} - k_i \cdot Y_{i-j}^{i-1}, j = 1, \dots, i-1$$

3) Result:

$$pDst[i-1] = Y_i^m, i = 1, \dots, m, pErr[0] = E^m.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> , <code>pDst</code> , or <code>pErr</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsNoOperation</code>	Indicates no solution to the LPC problem ($E^i \leq 0$).

Schur

*Calculates reflection coefficients
using Schur algorithm.*

Syntax

```
IppStatus ippsSchur_16s_sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp32f*
    pErr, int scaleFactor);
IppStatus ippsSchur_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f*
    pErr);
```

Parameters

<i>pSrc</i>	Pointer to the input autocorrelations vector [<i>len</i> +1].
<i>pDst</i>	Pointer to the output reflection coefficients vector [<i>len</i>].
<i>len</i>	Length of the output vector.
<i>pErr</i>	Pointer to the result prediction error.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippSchur` is declared in the `ippsr.h` file. This function uses Schur algorithm to compute reflection coefficients in the following steps:

1) Initialization:

$$b_1^0 = pSrc[0], \quad a_j^0 = b_{j+1}^0 = pSrc[j], \quad \text{for } j = 1, \dots, len-1, \quad a_{len}^0 = pSrc[len]$$

2) Iterations for $i = 1, \dots, len$:

$$k_i = -a_i^{i-1} / b_i^{i-1}$$

$$a_j^i = a_j^{i-1} + k_i b_j^{i-1}, \quad b_j^i = b_{j-1}^{i-1} + k_i a_{j-1}^{i-1}, \quad j = i+1, \dots, len$$

3) Result:

$$pDst[i-1] = k_i, \quad i = 1, \dots, len$$

$$pErr[0] = b_{len}^{len-1} + k_{len} \cdot a_{len}^{len-1}.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pDst</i> , or <i>pErr</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsNoOperation</code>	Indicates no solution to the LPC problem ($b_i^i = 0$).

LPToSpectrum

Calculates smoothed magnitude spectrum.

Syntax

```
IppStatus ippsLPToSpectrum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pDst,
    int order, Ipp32s val, int scaleFactor);

IppStatus ippsLPToSpectrum_32f(const Ipp32f* pSrc, int len, Ipp32f* pDst,
    int order, Ipp32f val);
```

Parameters

<i>pSrc</i>	Pointer to the input LPC coefficients vector [<i>len</i>].
<i>pDst</i>	Pointer to the output LP spectrum coefficients vector [2^{order}].
<i>len</i>	Number of LPC coefficients.
<i>order</i>	FFT order for spectrum calculation.
<i>val</i>	The value to add to spectrum.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsLPToSpectrum` is declared in the `ippsr.h` file. This function computes the first half of a linear prediction magnitude spectrum as follows:

$$pDst[k] = \frac{1}{\left| val - \sum_{i=1}^{len} pSrc[i-1] \cdot e^{-\frac{jik\pi}{N}} \right|}, \quad k = 0, \dots, N-1, \quad N = 2^{order}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or when it is greater or equal to $2^{order+1}$.
<code>ippStsFftOrderErr</code>	Indicates an error when the <i>order</i> value is incorrect.

<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>+Inf</code> and for integer operations is set to <code>IPP_MAX_16S</code> .
------------------------------	---

LPToCepstrum

*Calculates cepstrum coefficients
from linear prediction coefficients.*

Syntax

```
IppStatus ippSLPToCepstrum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippSLPToCepstrum_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

<code>pSrc</code>	Pointer to the linear prediction coefficients [<code>len</code>].
<code>pDst</code>	Pointer to the cepstrum coefficients [<code>len</code>].
<code>len</code>	Number of elements in the source and destination vectors.
<code>scaleFactor</code>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippSLPToCepstrum` is declared in the `ippsr.h` file. This function calculates the cepstrum coefficients from the linear prediction coefficients according to the following formula:

$$pDst[k] = - \left(pSrc[k] + \frac{1}{k+1} \sum_{i=1}^k (k-i+1) \cdot pSrc[i-1] \cdot pDst[k-i] \right), k = 0 \dots len-1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

CepstrumToLP

Calculates linear prediction coefficients from cepstrum coefficients.

Syntax

```
IppStatus ippsCepstrumToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsCepstrumToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the cepstrum coefficients [<i>len</i>].
<i>pDst</i>	Pointer to the linear prediction coefficients [<i>len</i>].
<i>len</i>	Number of elements in the source and destination vectors.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsCepstrumToLP` is declared in the `ippsr.h` file. This function calculates the linear prediction coefficients from the cepstrum coefficients according to the following formula:

$$pDst[k] = - \left(pSrc[k] + \frac{1}{k+1} \sum_{i=1}^k (k-i+1) \cdot pSrc[k-i] \cdot pDst[i-1] \right), k=0 \dots len-1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

LPToReflection

Calculates the linear prediction reflection coefficients from the linear prediction coefficients.

Syntax

```
IppStatus ippsLPToReflection_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
                                     int scaleFactor);
IppStatus ippsLPToReflection_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the linear prediction coefficients [<i>len</i>].
<i>pDst</i>	Pointer to the linear prediction reflection coefficients [<i>len</i>].
<i>len</i>	Number of elements in the source and destination vectors.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsLPToReflection` is declared in the `ippsr.h` file. This function calculates the linear prediction reflection coefficients according to the following formulae:

1) Initialization:

$$n = len, \quad a_i^n = pSrc[i-1], \quad i = 1, \dots, n$$

2) Iterations for $i = n, \dots, 1$:

$$k_i = a_i^i, \quad a_j^{i-1} = \frac{a_j^i - a_i^i \cdot a_{i-j}^i}{1 - k_i^2}, \quad j = 1, \dots, i-1$$

3) Result:

$$pDst[i-1] = k_i, \quad i = 1, \dots, n.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

`ippStsNoOperation` Indicates that reflection coefficients could not be calculated (that is, $|k_i| = 1$ for one of iterations).

ReflectionToLP

Calculates the linear prediction coefficients from the linear prediction reflection coefficients.

Syntax

```
IppStatus ippReflectionToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippReflectionToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

<code>pSrc</code>	Pointer to the linear prediction reflection coefficients [<code>len</code>].
<code>pDst</code>	Pointer to the linear prediction coefficients [<code>len</code>].
<code>len</code>	Number of elements in the source and destination vectors.
<code>scaleFactor</code>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippReflectionToLP` is declared in the `ippsr.h` file. This function calculates the linear prediction reflection coefficients from the linear prediction coefficients according to the following formulae:

1) Initialization:

$$n = len, \quad k_i = pSrc[i-1], \quad i = 1, \dots, n$$

2) Iterations for $i = 1, \dots, m$:

$$a_i^i = k_i, \quad a_j^i = a_j^{i-1} - k_i \cdot a_{i-j}^{i-1}, \quad j = 1, \dots, i-1$$

3) Result:

$$pDst[i-1] = a_i^n, \quad i = 1, \dots, n.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

ReflectionToAR

Converts reflection coefficients to area ratios.

Syntax

```

IppStatus ippsReflectionToAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsReflectionToLAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
    Ipp16s* pDst, int len, Ipp32f val, int scaleFactor);
IppStatus ippsReflectionToLAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f val);
IppStatus ippsReflectionToTrueAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToTrueAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len);

```

Parameters

<code>pSrc</code>	Pointer to the input vector [<code>len</code>].
<code>pDst</code>	Pointer to the destination vector [<code>len</code>].
<code>len</code>	Length of the input and output vectors.
<code>val</code>	Threshold value ($1 > val > 0$).
<code>srcShiftVal</code>	Scale factor used in <code>ippsReflectionToLAR</code> function only. Refer to “Integer Scaling” in Chapter 2.
<code>scaleFactor</code>	Refer to “Integer Scaling” in Chapter 2.

Description

These functions are declared in the `ippsr.h` file.

The function `ippsReflectionToAR` calculates approximate area ratios using the following formula:

$$pDst[k] = \frac{1 - pSrc[k]}{1 + pSrc[k]}, \quad k = 0, \dots, len-1$$

The function `ippsReflectionToLAR` calculates logarithm of area ratios as given by:

$$pDst[k] = \begin{cases} \ln \frac{1 - val}{1 + val}, & \text{if } pSrc[k] \geq val \\ \ln \frac{1 - pSrc[k]}{1 + pSrc[k]}, & \text{if } val > pSrc[k] > -val, \quad k = 0, \dots, len-1 \\ \ln \frac{1 + val}{1 - val}, & \text{if } -val \geq pSrc[k] \end{cases}$$

The function `ippsReflectionToTrueAR` calculates area ratios using the formulae:

$$pDst[0] = \frac{1 - pSrc[0]}{1 + pSrc[0]}, \quad pDst[k] = pDst[k-1] \cdot \frac{1 - pSrc[k]}{1 + pSrc[k]}, \quad k = 1, \dots, len-1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to: <div style="margin-left: 40px;"> <code>NaN</code> , if the dividend vector element has zero value; <code>+Inf</code> , if the dividend vector element has positive value; <code>-Inf</code> , if the dividend vector element is negative. </div> The value of the destination vector element for integer operations is set to: <div style="margin-left: 40px;"> <code>IPP_MAX_16S</code> , if the dividend vector element is positive; <code>IPP_MIN_16S</code> , if the dividend vector element is negative. </div>
<code>ippStsRangeErr</code>	Indicates an error when the condition <code>1 > val > 0</code> is not met.

ReflectionToTilt

Calculates tilt for rise/fall/connection parameters.

Syntax

```
IppStatus ippsReflectionToAbsTilt_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToAbsTilt_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsReflectionToTilt_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToTilt_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
```

Parameters

<i>pSrc1</i>	Pointer to the first input vector [<i>len</i>].
<i>pSrc2</i>	Pointer to the second input vector [<i>len</i>].
<i>pDst</i>	Pointer to the destination vector [<i>len</i>].
<i>len</i>	Length of the input and output vectors.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

These functions are declared in the `ippsr.h` file.

The function `ippsReflectionToAbsTilt` converts rise and fall coefficients to absolute tilt as given by:

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{|pSrc1[k]| + |pSrc2[k]|}, \quad k = 0, \dots, len-1.$$

The function `ippsReflectionToTilt` converts rise and fall coefficients to tilt:

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{pSrc1[k] + pSrc2[k]}, \quad k = 0, \dots, len-1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> or <i>pDst</i> pointer is NULL.

<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to: <div style="margin-left: 40px;"> <code>NaN</code> , if the dividend vector element has zero value; <code>+Inf</code> , if the dividend vector element has positive value; <code>-Inf</code> , if the dividend vector element is negative. </div> The value of the destination vector element for integer operations is set to: <div style="margin-left: 40px;"> <code>IPP_MAX_16S</code> , if the dividend vector element is positive; <code>IPP_MIN_16S</code> , if the dividend vector element is negative. </div>

PitchmarkToF0

*Calculates rise and fall amplitude
and duration for tilt.*

Syntax

```
IppStatus ippSPitchmarkToF0Cand_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, int scaleFactor);
IppStatus ippSPitchmarkToF0Cand_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len</i>].
<i>pDst</i>	Pointer to the destination vector [<i>len</i>].
<i>len</i>	Length of the input and output vectors.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippSPitchmarkToF0Cand` is declared in the `ippsr.h` file. This function calculates F0 candidate values from pitchmarks using the following formula:

$$pDst[0] = \frac{1}{pSrc[0]}, \quad pDst[k] = \frac{1}{pSrc[k] - pSrc[k-1]}, \quad k = 1, \dots, len-1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to +Inf. The value of the destination vector element for integer operations is set to IPP_MAX_16S.

UnitCurve

Calculates tilt for rise and fall coefficients.

Syntax

```

IppStatus ippsUnitCurve_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsUnitCurve_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsUnitCurve_16s_ISfs(Ipp16s* pSrcDst, int srcShiftVal, int len, int
    scaleFactor);
IppStatus ippsUnitCurve_32f_I(Ipp32f* pSrcDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len</i>].
<i>pSrcDst</i>	Pointer to the input and destination vector [<i>len</i>].
<i>pDst</i>	Pointer to the destination vector [<i>len</i>].
<i>len</i>	Length of the input and output vectors.
<i>srcShiftVal</i>	Refer to “Integer Scaling” in Chapter 2.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsUnitCurve` is declared in the `ippsr.h` file. This function calculates the destination vector according to the following formula:

$$pDst[k] = \begin{cases} 0, & \text{if } pSrc[k] < 0 \\ pSrc[k]^2, & \text{if } 0 \leq pSrc[k] < 1 \\ 2 - (2 - pSrc[k])^2, & \text{if } 1 \leq pSrc[k] \leq 2 \\ 2, & \text{if } 2 < pSrc[k] \end{cases}, \text{ for } k = 0, \dots, len-1.$$

In-place function flavors overwrite source vector with destination data.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pSrcDst</code> or <code>pDst</code> pointer is NULL.
<code>ippsStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

LPToLSP

Calculates line spectrum pairs vector from linear prediction coefficients.

Syntax

```
IppStatus ippsLPToLSP_16s_Sfs(const Ipp16s* pSrcLP, int srcShiftVal, Ipp16s* pDstLSP, int len, int* nRoots, int nInt, int nDiv, int scaleFactor);
IppStatus ippsLPToLSP_32f(const Ipp32f* pSrcLP, Ipp32f* pDstLSP, int len, int* nRoots, int nInt, int nDiv);
```

Parameters

<code>pSrcLP</code>	Pointer to the input LP coefficients vector [<code>len</code>].
<code>pDstLSP</code>	Pointer to the output LSP coefficients vector [<code>len</code>].
<code>len</code>	Number of LP coefficients.
<code>nRoots</code>	Pointer to the number of LSP values found.
<code>nInt</code>	Number of intervals in zero crossing search.

<i>nDiv</i>	Number of interval bisections during search for roots.
<i>srcShiftVal</i>	Scale factor for <i>pSrcLP</i> values.
<i>scaleFactor</i>	Scale factor for <i>pDstLSP</i> values, refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsLPToLSP` is declared in the `ippsr.h` file. This function converts linear prediction (LP) coefficients to the line spectrum pair (LSP) representation and implements the algorithm generally used in speech coding.

Let $a_k = pSrcLP[k]$, $k = 0, \dots, len-1$ (or $a_k = pSrcLP[k] \cdot 2^{-srcShiftVal}$ in case of the `ippsLPToLSP_16s_Sfs` function) be the LP coefficients (see also the [formula](#) used in the `ippsLinearPredictionNeg_Auto` function to compute LP coefficients), and

$$A(z) = 1 - \sum_{k=0}^{len-1} a_k z^{-k}.$$

LSP coefficients are defined as the roots of symmetric and anti-symmetric polynomials of the form

$$F_1(z) = A(z) + z^{-len-1} \cdot A(z^{-1}), \quad F_2(z) = A(z) - z^{-len-1} \cdot A(z^{-1}),$$

with the exception of the roots equal to 1 and -1, which are excluded through the definition of the following polynomials $G_1(z)$ and $G_2(z)$:

$$G_1(z) = F_1(z) / (1 + z^{-1}), \quad G_2(z) = F_2(z) / (1 - z^{-1}), \quad \text{if } len \text{ is even};$$

$$G_1(z) = F_1(z), \quad G_2(z) = F_2(z) / (1 - z^{-2}), \quad \text{if } len \text{ is odd}.$$

These polynomials are symmetric, so that

$$G_i(z) = g_{i, m_i} \cdot z^{-m_i} + \sum_{k=0}^{m_i-1} g_{i, k} \cdot (z^{-k} + z^{-(2m_i-k)}),$$

and they can be represented in the form

$$G_i(e^{jw}) = e^{-jwm_i} \cdot G_i'(w) = g_{i, m_i} + 2 \cdot \sum_{k=0}^{m_i-1} g_{i, k} \cdot T_{m_i-k}(x), \quad \text{for } i = 1, 2,$$

where $x = \cos w$, and $T_k(x) = \cos kw$ are the m -th order Chebyshev polynomials (see [Kab86]).

When len is even, the polynomial degrees are given by $m_1 = m_2 = len/2$, and

$g_{1,0} = g_{2,0} = 1$, $g_{1,k} = a_{k-1} + a_{len-k} - g_{1,k-1}$, $g_{2,k} = a_{k-1} - a_{len-k} + g_{2,k-1}$,
for $k = 1, \dots, m_1$.

When len is odd, $m_1 = (len+1)/2$, $m_2 = (len-1)/2$ and

$g_{1,0} = 1$, $g_{1,k} = -a_{k-1} - a_{len-k}$ for $k = 1, \dots, m_1$,

$g_{2,-1} = 0$, $g_{2,0} = 1$, and if $m_2 > 0$ then $g_{2,k} = g_{2,k-2} + a_{len-k} - a_{k-1}$ for $k = 1, \dots, m_2$.

Polynomials $G_1(z)$ and $G_2(z)$ have distinct interlaced roots on the unit circle that are found in cosine domain by checking for polynomials sign change at $nInt$ points equally spaced between 0 and π (zero crossing search). The sign change interval is then bisected $nDiv$ times and after that the root is located by linear interpolation. The Chebyshev polynomials are used to evaluate $G_1(z)$ and $G_2(z)$. The roots are written to `pDstLSP` vector in ascending order.

The number of roots found is written to `nRoots[0]`. If all len roots were not found, the error status is returned.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcLP</code> , <code>pDstLSP</code> or <code>nRoots</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when len is less than or equal to 0, or $nInt$ is less than len , or $nDiv$ is less than 0.
<code>ippStsNoRootFoundErr</code>	Indicates that less than len roots were found.

LSPToLP

*Converts line spectrum pairs vector
to linear prediction coefficients.*

Syntax

```
IppStatus ippsLSPToLP_16s_Sfs(const Ipp16s* pSrcLSP, int srcShiftVal, Ipp16s*
    pDstLP, int len, int scaleFactor);
IppStatus ippsLSPToLP_32f(const Ipp32f* pSrcLSP, Ipp32f* pDstLP, int len);
```

Parameters

<i>pSrcLSP</i>	Pointer to the input LSP coefficients vector [<i>len</i>].
<i>pDstLP</i>	Pointer to the output LP coefficients vector [<i>len</i>].
<i>len</i>	Number of LSP and LP coefficients.
<i>srcShiftVal</i>	Scale factor for <i>pSrcLSP</i> values.
<i>scaleFactor</i>	Scale factor for <i>pDstLP</i> values, refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsLSPToLP` is declared in the `ippsr.h` file. This function converts line spectrum pair (LSP) values to linear prediction (LP) coefficients using the Kabal’s method (see [\[Kab86\]](#)).

The polynomials

$$G_1(e^{-jw}) = e^{-jwm_1} \cdot G_1'(w) \quad \text{and} \quad G_2(e^{-jw}) = e^{-jwm_2} \cdot G_2'(w)$$

are reconstructed from their roots $pSrcLSP[k]$, $k = 0, \dots, len-1$, using Chebyshev representation.

Next the polynomials $F_1(z)$ and $F_2(z)$ and are reconstructed as:

$$F_1(z) = G_1(z) \cdot (1 + z^{-1}), \quad F_2(z) = G_2(z) \cdot (1 - z^{-1}), \quad \text{if } len \text{ is even};$$

$$F_1(z) = G_1(z), \quad F_2(z) = G_2(z) \cdot (1 - z^{-2}), \quad \text{if } len \text{ is odd}.$$

Finally, the linear prediction coefficients are determined from

$$A(z) = \frac{F_1(z) + F_2(z)}{2} = 1 - \sum_{k=0}^{len-1} a_k z^{-k}$$

and

$$pDstLSP[k] = a_k, k = 0, \dots, len-1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcLSP</code> or <code>pDstLSP</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsIncorrectLSP</code>	Indicates a warning that <code>pSrcLSP</code> elements are not valid LSP values (so that condition $-1 < \dots < pSrcLSP[k+1] < pSrcLSP[k] < \dots < 1$ is not met).

MelToLinear

*Converts Mel-scaled values
to linear scale values.*

Syntax

```
IppStatus ippMelToLinear_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
    melMul, Ipp32f melDiv);
```

Parameters

<code>pSrc</code>	Pointer to the input vector [<code>len</code>].
<code>pDst</code>	Pointer to the output vector [<code>len</code>].
<code>len</code>	Length of input and output vectors.
<code>melMul</code>	Multiply factor in the Mel-scale equation.
<code>melDiv</code>	Divide factor in the Mel-scale equation.

Description

The function `ippMelToLinear` is declared in the `ippsr.h` file. This function converts the Mel-frequency scale to the linear frequency scale:

$$pDst[i] = melDiv \cdot \left[\exp\left(\frac{pSrc[i]}{melMul}\right) - 1 \right], i = 0, \dots, len-1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0, or <code>melMul</code> or <code>melDiv</code> is equal to 0.

LinearToMel

Converts linear-scale values to Mel-scale values.

Syntax

```
IppStatus ippLinearToMel_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
    melMul, Ipp32f melDiv);
```

Parameters

<code>pSrc</code>	Pointer to the input vector [<code>len</code>].
<code>pDst</code>	Pointer to the output vector [<code>len</code>].
<code>len</code>	Length of the input and output vectors.
<code>melMul</code>	Multiply factor in the Mel-scale equation.
<code>melDiv</code>	Divide factor in the Mel-scale equation.

Description

The function `ippLinearToMel` is declared in the `ippsr.h` file. This function converts the linear frequency scale to the Mel-frequency scale:

$$pDst[i] = melMul \cdot \ln\left(1 + \frac{pSrc[i]}{melDiv}\right), \quad i = 0, \dots, len-1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0, or <code>melMul</code> or <code>melDiv</code> is equal to 0.

CopyWithPadding

Copies the input signal to the output with zero-padding.

Syntax

```
IppStatus ippsCopyWithPadding_16s(const Ipp16s* pSrc, int lenSrc, Ipp16s* pDst,
    int lenDst);
IppStatus ippsCopyWithPadding_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f* pDst,
    int lenDst);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>lenSrc</i>].
<i>pDst</i>	Pointer to the output vector [<i>lenDst</i>].
<i>lenSrc</i>	Length of the input vector <i>pSrc</i> .
<i>lenDst</i>	Length of the output vector <i>pDst</i> .

Description

The function `ippsCopyWithPadding` is declared in the `ippsr.h` file. This function copies the input vector to the output vector. If the length of the output vector is bigger, the trailing elements are padded with zeroes.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>lenSrc</i> or <i>lenDst</i> is less than or equal to 0, or when <i>lenDst</i> is less than <i>lenSrc</i> .

MelFBankGetSize

Gets the size of the Mel-frequency filter bank structure.

Syntax

```
IppStatus ippsMelFBankGetSize_32s(int winSize, int nFilter, IppMelMode mode,
    int* pSize);
```

Parameters

<i>winSize</i>	Frame length in samples ($32 \leq \textit{winSize} \leq 8192$).
<i>nFilter</i>	Number of Mel-scale filter banks K ($0 < \textit{nFilter} \leq \textit{winSize}$).
<i>mode</i>	Flag that determines the execution mode. Currently only IPP_FBANK_FREQWGT is supported.
<i>pSize</i>	Pointer to the variable to contain the size of the filter bank structure.

Description

The function `ippsMelFBankGetSize` is declared in the `ippsr.h` file. This function determines the size required for the Mel-frequency filter bank structure and associated storage. It should be called before memory allocation and before the call to `ippsMelFBankInit`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsFBankFlagErr</code>	Indicates an error when the <i>mode</i> value is incorrect.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>nFilter</i> is out of range or when <i>winSize</i> is less than or equal to 0.

MelFBankInit

Initializes the structure for performing the Mel-frequency filter bank analysis.

Syntax

```
IppStatus ippsMelFBankInit_32s(IppsFBankState_32s* pFBank, int* pFFTLen, int
    winSize, Ipp32s sampFreq, Ipp32s lowFreq, Ipp32s highFreq, int nFilter,
    Ipp32s melMulQ15, Ipp32s melDivQ15, IppMelMode mode);
```

Parameters

<i>pFBank</i>	Pointer to the Mel-scale filter bank structure to be initialised.
<i>pFFTLen</i>	Pointer to the length of FFT ($N = pFFTLen[0]$) used for the filter bank evaluation.
<i>winSize</i>	Frame length (in samples).
<i>sampFreq</i>	Input signal sampling frequency f_i (in Hz).
<i>lowFreq</i>	Start frequency f_{low} of the first band-pass filter (in Hz).
<i>highFreq</i>	End frequency f_{high} of the last band-pass filter (in Hz).
<i>nFilter</i>	Number of Mel-scale filter banks K.
<i>melMulQ15</i>	Real-valued mel-scale formula multiply factor in Q15.
<i>melDivQ15</i>	Real-valued mel-scale formula divisor in Q15.
<i>mode</i>	Flags that determine execution mode; can have the following values: <div style="margin-left: 40px;"> IPP_FBANK_MELWGT – the function calculates filter bank weights in Mel-scale; IPP_FBANK_FREQWGT – the function calculates filter bank weights in the frequency space. One of the above two flags should necessarily be set. IPP_POWER_SPECTRUM – indicates that the FFT power spectrum is used during the filter bank analysis. </div>

Description

The function `ippsMelFBankInit` is declared in the `ippsr.h` file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis function. Mel filter coefficients are calculated in the same way as for the `ippsMelFBankInitAlloc` functions. Memory for the structure should be previously allocated, the size of this memory can be determined by the `ippsMelFBankGetSize` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pFBank</code> or <code>pFFTLen</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>winSize</code> , <code>nFilter</code> , <code>sampFreq</code> , or <code>lowFreq</code> is less than or equal to 0.
<code>ippStsFBankFreqErr</code>	Indicates an error when <code>highFreq</code> is less than <code>lowFreq</code> or <code>highFreq</code> is greater than <code>sampFreq/2</code> .
<code>ippStsFBankFlagErr</code>	Indicates an error when the <code>mode</code> value is incorrect.

MelFBankInitAlloc

Allocates memory and initializes the structure for performing the Mel-frequency filter bank analysis.

Syntax

```
IppStatus ippsMelFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f highFreq,
    int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);

IppStatus ippsMelFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f highFreq,
    int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

Parameters

<code>pFBank</code>	Pointer to the Mel-scale filter bank structure to be created.
<code>pFFTLen</code>	Pointer to the length of FFT ($N = pFFTLen[0]$) used for the filter bank evaluation.
<code>winSize</code>	Frame length (in samples).

<i>sampFreq</i>	Input signal sampling frequency f_i (in Hz).
<i>lowFreq</i>	Start frequency f_{low} of the first band-pass filter (in Hz).
<i>highFreq</i>	End frequency f_{high} of the last band-pass filter (in Hz).
<i>nFilter</i>	Number of Mel-scale filter banks K .
<i>melMul</i>	Mel-scale formula multiply factor.
<i>melDiv</i>	Mel-scale formula divisor.
<i>mode</i>	Flags that determine execution mode; can have the following values: <ul style="list-style-type: none"> <code>IPP_FBank_MELWGT</code> – the function calculates filter bank weights in Mel-scale; <code>IPP_FBank_FREQWGT</code> – the function calculates filter bank weights in the frequency space. <p>One of the above two flags should necessarily be set.</p> <code>IPP_POWER_SPECTRUM</code> – indicates that the FFT power spectrum is used during the filter bank analysis.

Description

The function `ippsMelFBankInitAlloc` is declared in the `ippsr.h` file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis. The filter bank analysis is one of the major steps in the Mel-Frequency Cepstrum Coefficients (MFCC) feature calculation.

The Mel-frequency translation is accomplished according to the following equation:

$$mel(f) = melMul \cdot \ln\left(1 + \frac{f}{melDiv}\right)$$

The center of each filter bank (c_k in Mel-scale and y_k in FFT domain) is calculated as follows:

$$c_k = mel(f_{low}) + \frac{k}{K+1} \cdot [mel(f_{high}) - mel(f_{low})],$$

$$y_k = round\left\{\frac{N}{f_s} mel^{-1}(c_k)\right\},$$

for $k = 0 \dots K+1$,

where $round(x)$ rounds the value of x to the nearest integer (when *mode* is equal to `IPP_FBank_FREQWGT`) or to the largest integer that is less than or equal to x (when *mode* is equal to `IPP_FBank_MELWGT`), and N is the length of FFT.

If *mode* is IPP_FBANK_FREQWGT, then the filter outputs will be calculated (by the function [ippsEvalFBank](#)) according to the following formula:

$$fFank_{k-1} = \sum_{i=y_{k-1}}^{y_k} \frac{i - y_{k-1} + 1}{y_k - y_{k-1} + 1} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{y_{k+1} - i + 1}{y_{k+1} - y_k + 1} \cdot x_i, \text{ for } k = 1 \dots K, \quad (8.1)$$

where x_i is the magnitude of the corresponding FFT spectrums.

If *mode* is IPP_FBANK_MELWGT, the filter outputs will be calculated (by the function [ippsEvalFBank](#)) according to the formula:

$$fFank_{k-1} = \sum_{i=y_{k-1}+1}^{y_k} \frac{\text{mel}\left(i \frac{f_s}{N}\right) - c_{k-1}}{c_k - c_{k-1}} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{c_{k+1} - \text{mel}\left(i \frac{f_s}{N}\right)}{c_{k+1} - c_k} \cdot x_i, k = 1 \dots K \quad (8.2)$$

For the function flavor `ippsMelFBankInitAlloc_32s`, currently only IPP_FBANK_MELWGT mode is supported.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pFBank</i> or <i>pFFTLen</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>winSize</i> , <i>nFilter</i> , <i>sampFreq</i> , or <i>lowFreq</i> is less than or equal to 0.
<code>ippStsFBankFreqErr</code>	Indicates an error when <i>highFreq</i> is less than <i>lowFreq</i> or <i>highFreq</i> is greater than <i>sampFreq</i> /2.
<code>ippStsFBankFlagErr</code>	Indicates an error when the <i>mode</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.

MelLinFBankInitAlloc

Initializes the structure for performing a combined linear and Mel-frequency filter bank analysis.

Syntax

```
IppStatus ippsMelLinFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
    Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
    Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);

IppStatus ippsMelLinFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
    Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
    Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

Parameters

<i>pFBank</i>	Pointer to the filter bank structure to be created.				
<i>pFFTLen</i>	Pointer to the length of FFT ($N = pFFTLen[0]$) used for the filter bank evaluation.				
<i>winSize</i>	Frame length (in samples).				
<i>sampFreq</i>	Input signal sampling frequency f_i (in Hz).				
<i>lowFreq</i>	Start frequency f_{low} of the first band-pass filter (in Hz).				
<i>highLinFreq</i>	End frequency f_{lin} of the last band-pass filter in linear scale (in Hz).				
<i>highFreq</i>	End frequency f_{high} of the last band-pass filter (in Hz).				
<i>nFilter</i>	Number of Mel-scale filter banks K .				
<i>nLinFilter</i>	Number of linear-scale filter banks L .				
<i>melMul</i>	Mel-scale formula multiply factor.				
<i>melDiv</i>	Mel-scale formula divisor.				
<i>mode</i>	Flags determining the function's execution mode; can have the following values: <table> <tr> <td>IPP_FBANK_MELWGT</td><td>– the function calculates filter bank weights in Mel-scale;</td></tr> <tr> <td>IPP_FBANK_FREQWGT</td><td>– the function calculates filter bank weights in the frequency space.</td></tr> </table>	IPP_FBANK_MELWGT	– the function calculates filter bank weights in Mel-scale;	IPP_FBANK_FREQWGT	– the function calculates filter bank weights in the frequency space.
IPP_FBANK_MELWGT	– the function calculates filter bank weights in Mel-scale;				
IPP_FBANK_FREQWGT	– the function calculates filter bank weights in the frequency space.				

One of the above two flags should necessarily be set.

IPP_POWER_SPECTRUM – indicates that the FFT power spectrum is used during the filter bank analysis.

Description

The function `ippsMelLinFBankInitAlloc` is declared in the `ippsr.h` file. This function initializes the linear and Mel-frequency triangular filter banks. The first L filters are placed linearly on the frequency band from f_{low} to f_{lin} . The center of each filter bank (c_k in Mel-scale and y_k in FFT domain) is calculated as follows:

If $L = K$, then

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L + 1} \cdot k, \quad y_k = \text{round}\left\{\frac{N}{f_s} \cdot z_k\right\}, \quad c_k = \text{mel}(z_k), \quad k = 0, \dots, L+1$$

If $L < K$, then

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L} \cdot k, \quad y_k = \text{round}\left\{\frac{N}{f_s} \cdot z_k\right\}, \quad c_k = \text{mel}(z_k), \quad k = 0, \dots, L$$

and

$$c_k = \text{mel}(f_{lin}) + \frac{k - L}{K - L + 1} \cdot [\text{mel}(f_{high}) - \text{mel}(f_{lin})],$$

$$y_k = \text{round}\left\{\frac{N}{f_s} \cdot \text{mel}^{-1}(c_k)\right\}, \quad \text{for } k = L, \dots, K+1,$$

where $\text{round}(x)$ rounds the value of x to the nearest integer (when `mode` is equal to `IPP_FBANK_FREQWGT`) or to the largest integer that is less than or equal to x (when `mode` is `IPP_FBANK_MELWGT`), and N is the length of FFT.

If `mode` is `IPP_FBANK_FREQWGT`, the filter outputs are calculated (by the function [ippsEvalFBank](#)) according to the formula (8.1) given for the `ippsMelFBankInitAlloc` function.

If `mode` is `IPP_FBANK_MELWGT`, the filter output is calculated (by the function `ippsEvalFBank`) according to the formula (8.2).

Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when <i>pFBank</i> or <i>pFFTLen</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>winSize</i> , <i>nFilter</i> , <i>sampFreq</i> , <i>lowFreq</i> is less than or equal to 0, or when <i>nLinFilter</i> is less than or equal to 1, or <i>nLinFilter</i> is greater than <i>nFilter</i> .
<code>ippStsFBankFreqErr</code>	Indicates an error when <i>highLinFreq</i> is less than <i>lowFreq</i> , or <i>highFreq</i> is less than <i>highLinFreq</i> , or <i>highFreq</i> is greater than <i>sampFreq</i> /2, or $(highLinFreq > lowFreq) \&\& (nLinFilter == 0)$, or $(highLinFreq < highFreq) \&\& (nLinFilter == nFilter)$.
<code>ippStsFBankFlagErr</code>	Indicates an error when the <i>mode</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.

EmptyFBankInitAlloc

Initializes an empty filter bank structure.

Syntax

```
IppStatus ippsEmptyFBankInitAlloc_16s(IppsFBankState_16s** pFBank, int*
    pFFTLen, int winSize, int nFilter, IppMelMode mode);
IppStatus ippsEmptyFBankInitAlloc_32f(IppsFBankState_32f** pFBank, int*
    pFFTLen, int winSize, int nFilter, IppMelMode mode);
```

Parameters

<i>pFBank</i>	Pointer to the filter bank structure to be created.
<i>pFFTLen</i>	Pointer to the length of FFT ($N = pFFTLen[0]$) used for the filter bank evaluation.
<i>winSize</i>	Frame length (in samples).
<i>nFilter</i>	Number of filters banks K .
<i>mode</i>	Flag determining the function's execution mode; can have the following value: <div style="text-align: right;"> <code>IPP_POWER_SPECTRUM</code> – indicates that the FFT power spectrum is used during the filter bank analysis. </div>

Description

The function `ippsEmptyFBankInitAlloc` is declared in the `ippsr.h` file. This function initializes the filter bank structure for the *nFilter* filters. The filter bank center frequencies can be set by the function `ippsFBankSetCenters` and the filter weights can be set by the function `ippsFBankSetCoeffs`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pFBank</i> or <i>pFFTLen</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>winSize</i> or <i>nFilter</i> or <i>pFBank</i> is less than or equal to 0.

`ippStsMemAllocErr` Indicates an error when no memory was allocated.

FBankFree

Destroys the structure for the filter bank analysis.

Syntax

```
IppStatus ippSFBankFree_16s(IppsFBankState_16s* pFBank);  
IppStatus ippSFBankFree_32f(IppsFBankState_32f* pFBank);
```

Parameters

pFBank Pointer to the filter bank structure.

Description

The function `ippSFBankFree` is declared in the `ippsr.h` file. This function destroys the filter bank structure and frees all memory associated with it.

Return Values

`ippStsNoErr` Indicates no error.
`ippStsNullPtrErr` Indicates an error when *pFBank* pointer is NULL.

FBankGetCenters

Retrieves the center frequencies of the triangular filter banks.

Syntax

```
IppStatus ippSFBankGetCenters_16s(const IppsFBankState_16s* pFBank, int*  
    pCenters);  
IppStatus ippSFBankGetCenters_32f(const IppsFBankState_32f* pFBank, int*  
    pCenters);
```


Parameters

<i>pFBank</i>	Pointer to the filter bank structure.
<i>pCenters</i>	Pointer to the output vector that contains the center frequencies.

Description

The function `ippsGetCenters` is declared in the `ippsr.h` file. This function retrieves the indexes y_k (in FFT domain points) of the filter bank centers. The resulting array *pCenters* is of length $nFilter+2$. The filter bank structure *pFBank* must be initialized by either `ippsMelFBankInitAlloc` or `ippsMelLinFBankInitAlloc` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pFBank</i> or <i>pCenters</i> pointer is NULL.
<code>ippStsFBankErr</code>	Indicates an error when filter centers are not valid after filter bank initialization by <code>ippsEmptyFBankInitAlloc</code> function.

FBankSetCenters

Sets the center frequencies of the triangular filter banks.

Syntax

```
IppStatus ippsFBankSetCenters_16s(IppsFBankState_16s* pFBank, const int*
    pCenters);
IppStatus ippsFBankSetCenters_32f(IppsFBankState_32f* pFBank, const int*
    pCenters);
```

Parameters

<i>pFBank</i>	Pointer to the filter bank structure.
<i>pCenters</i>	Pointer to the vector that contains center frequencies.

Description

The function `ippsSetCenters` is declared in the `ippsr.h` file. This function sets the filter center indexes y_k in `pFBank`.

The indexes must meet the following conditions:

$$0 \leq y_k \leq y_{k+1} \leq N/2, \quad i = 0, \dots, K.$$

If the k -th center is modified, the filter weight coefficients in the $(k-1)$ -th, k -th and $(k+1)$ -th filter banks must be also modified using the `ippsFBankSetCoeffs` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pFBank</code> or <code>pCenters</code> pointer is NULL.

FBankGetCoeffs

Retrieves the filter bank weight coefficients.

Syntax

```
IppStatus ippsFBankGetCoeffs_16s(const IppsFBankState_16s* pFBank, int fIdx,
    Ipp32f* pCoeffs);
IppStatus ippsFBankGetCoeffs_32f(const IppsFBankState_32f* pFBank, int fIdx,
    Ipp32f* pCoeffs);
```

Parameters

<code>pFBank</code>	Pointer to the filter bank structure
<code>fIdx</code>	Filter index.
<code>pCoeffs</code>	Pointer to the output vector that contains the filter coefficients.

Description

The function `ippsFBankGetCoeffs` is declared in the `ippsr.h` file. This function copies the weight coefficients of the filter bank `fIdx` to the array `pCoeffs` which is of length $(y_{k+1} - y_{k-1} + 1)$.

Note that for `ippsFBankGetCoeffs_16s` function flavor, the filter output weights are represented in floating-point format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pFBank</code> or <code>pCoeffs</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>fIdx</code> is less then 1 or greater then <code>nFilter</code> .
<code>ippStsFBankErr</code>	Indicates an error when <code>fIdx</code> filter coefficients are not available or valid.

FBankSetCoeffs

Sets the filter bank weight coefficients.

Syntax

```
IppStatus ippsFBankSetCoeffs_16s(IppsFBankState_16s* pFBank, int fIdx, const
    Ipp32f* pCoeffs);
IppStatus ippsFBankSetCoeffs_32f(IppsFBankState_32f* pFBank, int fIdx, const
    Ipp32f* pCoeffs);
```

Parameters

<code>pFBank</code>	Pointer to the filter bank structure.
<code>fIdx</code>	Filter index.
<code>pCoeffs</code>	Pointer to the output coefficients vector.

Description

The function `ippsFBankGetCoeffs` is declared in the `ippsr.h` file. This function sets the weight coefficients of the filter bank `fIdx`. The vector `pCoeffs` contains the weight coefficients from y_{k-1} to y_{k+1} .

Note that for `ippsFBankSetCoeffs_16s` function flavor, the weight coefficients are represented in floating-point format and are saturated to the $[-1,1]$ interval.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pFBank</code> or <code>pCoeffs</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>fIdx</code> is less than 1 or greater than <code>nFilter</code> .
<code>ippStsFBankErr</code>	Indicates an error when the weight coefficients are not available or valid.

EvalFBank

Performs the filter bank analysis.

Syntax

```
IppStatus ippsEvalFBank_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pDst, const
    IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFBankState_32s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFBankState_32f* pFBank);
```

Parameters

<code>pSrc</code>	Pointer to the source vector $[2^{pFFTOrder[0]}]$.
<code>pDst</code>	Pointer to the filter bank coefficients vector $[nFilter]$.
<code>pFBank</code>	Pointer to the filter bank structure.
<code>scaleFactor</code>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsEvalFBank` is declared in the `ippsr.h` file. This function performs the filter bank analysis for the input vector `pSrc`.

The execution mode set during the filter bank structure initialization determines the use of the input vector as follows:

If the `IPP_POWER_SPECTRUM` flag is set, the source vector is the wave signals. The magnitude of input signal spectrum is calculated for the filter bank analysis as follows:

$$x_k = \left| \sum_{i=0}^{N-1} pSrc[i] \cdot \exp\left(-j \cdot 2\pi \frac{ik}{N}\right) \right|, \quad 0 \leq k \leq N/2.$$

Otherwise, the input vector is used directly for filter bank analysis:

$$x_k = pSrc[i], \quad 0 \leq k \leq N/2.$$

Depending on the mode, either formula (8.1) or (8.2) is used to obtain the filter bank coefficients. The input vector `pSrc` is destroyed after the analysis.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsFBankErr</code>	Indicates an error when <code>pFBank</code> structure is not ready for calculation.

DCTLifterGetSize_MulC0

Gets the size of the DCT structure.

Syntax

```
IppStatus ippSDCTLifterGetSize_MulC0_16s(int lenDCT, int lenCeps, int* pSize);
```

Parameters

<code>lenDCT</code>	Length of the DCT ($0 < lenDCT \leq 192$).
<code>lenCeps</code>	Number of the output coefficients not including C0 ($0 < lenCeps \leq lenDCT$).
<code>pSize</code>	Pointer to the variable to contain the size in bytes.

Description

The function `ippsDCTLifterGetSize_MulC0` is declared in the `ippsr.h` file. This function computes the size in bytes required for the DCT structure and associated storage.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>lenDCT</i> or <i>lenCeps</i> are out of bounds.

DCTLifterInit_MulC0

Initializes the structure to perform DCT and lift the DCT coefficients.

Syntax

```
IppStatus ippsDCTLifterInit_MulC0_16s(IppsDCTLifterState_16s* pDCTLifter, int
    lenDCT, const Ipp32s* pLifterQ15, int lenCeps);
```

Parameters

<i>pDCTLifter</i>	Pointer to the structure to be initialized for the DCT calculation and liftering.
<i>lenDCT</i>	Length of the DCT ($0 < lenDCT \leq 8192$).
<i>pLifterQ15</i>	Pointer to the Q15 format liftering coefficients vector ($0.0_{Q15} \leq pLifterQ15[k] < 512.0_{Q15}$).
<i>lenCeps</i>	Number of the output coefficients not including C_0 ($0 < lenCeps \leq lenDCT$).

Description

The function `ippsDCTLifterInit_MulC0` is declared in the `ippsr.h` file. This function initializes the structure for the DCT calculation and liftering similar to what is done by the `ippsDCTLifterInitAlloc_MulC0` function. However, memory for the structure should be previously allocated, and the size of memory could be determined by the `ippsDCTLifterGetSize_MulC0` function.

C_0 is stored as the last element of the output vector.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pLifterQ15</code> or <code>pDCTLifter</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>lenDCT</code> , <code>lenCeps</code> , or <code>pLifterQ15[k]</code> is out of range.

DCTLifterInitAlloc

Initializes the structure and allocates memory to perform DCT and lift the DCT coefficients.

Syntax

```

IppStatus ippsDCTLifterInitAlloc_16s(IppsDCTLifterState_16s** pDCTLifter,
    int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_16s(IppsDCTLifterState_16s** pDCTLifter,
    int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_16s(IppsDCTLifterState_16s** pDCTLifter,
    int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_16s(IppsDCTLifterState_16s** pDCTLifter,
    int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_32f(IppsDCTLifterState_32f** pDCTLifter,
    int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_32f(IppsDCTLifterState_32f** pDCTLifter,
    int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_32f(IppsDCTLifterState_32f** pDCTLifter,
    int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_32f(IppsDCTLifterState_32f** pDCTLifter,
    int lenDCT, const Ipp32f* pLifter, int lenCeps);

```

Parameters

<code>pDCTLifter</code>	Pointer to the structure to be created for the DCT calculation and lifting.
-------------------------	---

<i>lenDCT</i>	Length of the DCT.
<i>lenCeps</i>	Number of the output coefficients (not including C_0).
<i>nLifter</i>	Liftering factor.
<i>pLifter</i>	Pointer to the liftering coefficients vector.
<i>val</i>	The scale factor for the output coefficients (except C_0).
<i>val0</i>	The scale factor for C_0 .

Description

The function `ippSDCTLifterInitAlloc` is declared in the `ippsr.h` file. This function initializes the structure for the DCT calculation and liftering.

The first DCT coefficient C_0 is usually ignored. Therefore, the output of the `ippSDCTLifter` function contains only C_1 to $C_{lenCeps}$ coefficients. However, if the `C0` suffix is specified, C_0 is stored as the last element of the output vector.

The liftering coefficients are calculated according to the below formulae:

For functions without both the `Mu1` and `C0` suffixes,

$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val, \quad i = 1, \dots, lenCeps$$

For functions with the `C0` suffix,

$$l_0 = val0$$

$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val, \quad i = 1, \dots, lenCeps$$

For functions with the `Mu1` suffix,

$$l_i = pLifter[i-1], \quad i = 1, \dots, lenCeps$$

For functions that have both the `Mu1` and `C0` suffixes:

$$l_0 = pLifter[lenCeps-1]$$

$$l_i = pLifter[i-1], \quad i = 1, \dots, lenCeps$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pLifter</code> pointer is <code>NULL</code> .

<code>ippStsSizeErr</code>	Indicates an error when <i>lenDCT</i> , <i>lenCeps</i> , or <i>nLifter</i> is less than or equal to 0, or when <i>lenDCT</i> is less than <i>lenCeps</i> .
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.

DCTLifterFree

Destroys the structure used for the DCT and liftering.

Syntax

```
IppStatus ippSDCTLifterFree_16s(IppsDCTLifterState_16s* pDCTLifter);
IppStatus ippSDCTLifterFree_32f(IppsDCTLifterState_32f* pDCTLifter);
```

Parameters

pDCTLifter Pointer to the DCT and liftering structure.

Description

The function `ippSDCTLifterFree` is declared in the `ippsr.h` file. This function closes the DCT and liftering structure by freeing all memory associated with it.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDCTLifter</i> pointer is NULL.

DCTLifter

Performs the DCT and lifts the DCT coefficients.

Syntax

```
IppStatus ippSDCTLifter_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
IppStatus ippSDCTLifter_32s16s_Sfs (const Ipp32s* pSrc, Ipp16s* pDst, const
    IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
```

```
IppStatus ippsDCTLifter_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDCTLifterState_32f* pDCTLifter);
```

Parameters

<i>pSrc</i>	Pointer to the source vector [<i>lenDCT</i>].
<i>pDst</i>	Pointer to the output vector [<i>lenCeps</i>] or [<i>lenCeps</i> +1].
<i>pDCTLifter</i>	Pointer to the DCT and liftering structure.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsDCTLifter` is declared in the `ippsr.h` file. This function first performs the DCT and then lifts the DCT coefficients.

The DCT coefficients are calculated according to the formula:

$$y_i = \sum_{j=1}^{lenDCT} pSrc[j-1] \cdot \cos\left(\frac{\pi \cdot i \cdot (j-0.5)}{lenDCT}\right), \quad i = 0, \dots, lenCeps$$

The output coefficients are weighted as follows:

$$pDst[i-1] = l_i \cdot y_i, \quad i = 1, \dots, lenCeps$$

If the C_0 coefficient is required (*pDCTLifter* is initialized by functions with the `C0` suffix), it is stored as the last element of the output vector:

$$pDst[lenCeps] = l_0 \cdot y_0$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pDst</i> , or <i>pFBank</i> pointer is NULL.

The following example shows how the feature processing functions can be used for MFCC feature calculation.

Example 8-1 MFCC feature calculation

```
/* Input: samples[] Input samples
sample_number Number of samples
```

Output: mfccs[12*(sample_num-240)/160] The resulting MFCC coefficients
*/

```
void Calc_MFCC (Ipp32f *samples, int sample_num, Ipp32f *mfccs) {
    Ipp32f* frame_buffer,fbank_buffer,mfcc_cur;
    IppsFBankState_32f *fbank;
    IppsDCTLifterState_32f *dctl;
    int fft_len,fft_order;
    int i,j;
    float LogEnergy;

    /* Initialize the structures */
    ippsMelFBankInitAlloc_32f(&fbank, /* return the structure pointer */
        &fft_order, /* return the FFT length */
        400, /* 25ms window/512 point FFT */
        16000, /* sample rate */
        64, /* lowest frequency of interest */
        8000, /* highest frequency of interest */
        24, /* number of filter banks */
        1127.0, /* mel-scale factor 1 */
        700.0, /* mel-scale factor 2 */
        IPP_FBANK_MELWGT | IPP_POWER_SPECTRUM);
    ippsDCTLifterInitAlloc_32f(&dctl, /* return the structure pointer */
        24, /* filter bank channels */
        12, /* number of MFCC coefficients */
        22, /* liftering */
        1.0); /* no scaling */

    fft_len=1<<fft_order;
    frame_buffer=ippsMalloc_32f(fft_len);
    fbank_buffer=ippsMalloc_32f(24);
    mfcc_cur=mfccs;

    /* Calculate MFCC features */
    for (i=j=0; i+400<sample_num; i+=160,mfcc_cur+=12,j++) {
        /* Organize the input wave data into a frame */
        ippsCopyWithPadding_32f(&samples[i],400,frame_buffer,fft_len);
        ippsDotProd_32f(frame_buffer,frame_buffer,fft_len,&LogEnergy);
        /* Pre-emphasize the input signal with factor 0.97 */
        ippsPreemphasize_32f(frame_buffer,400,0.97);
        frame_buffer[0]*=(1.0-0.97);
    }
}
```

```

    /* Add the hamming window to the input signal */
    ippsWinHamming_32f_I(frame_buffer,400);
    /* Perform the filter bank analysis */
    ippsEvalFBank_32f(frame_buffer,fbank_buffer,fbank);
    ippsThreshold_LTVal_32f_I(fbank_buffer, 24, 1.0, 1.0);
    ippsLn_32f_I(fbank_buffer, 24);
    /* Perform the DCT analysis and liftering */
    ippsDCTLifter_32f(fbank_buffer,mfcc_cur,dctl);
    mfcc_cur[12] = (float) log ((double) LogEnergy);

}
/* Normalize log energy */
ippsNormEnergy_32f(mfccs+11,12,(sample_num-240)/160,50.0,1.0);
/* Destroy the structures after calculation */
ippsFree(fbank_buffer);
ippsFree(frame_buffer);
ippsFBankFree_32f(fbank);
ippsDCTLifterFree_32f(dctl);
}

```

NormEnergy

Normalizes a vector of energy values.

Syntax

```

IppStatus ippsNormEnergy_32f(Ipp32f* pSrcDst, int step, int height,
    Ipp32f silFloor, Ipp32f enScale);
IppStatus ippsNormEnergy_16s(Ipp16s* pSrcDst, int step, int height,
    Ipp16s silFloor, Ipp16s val, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_32f(Ipp32f* pSrcDst, int step, int height,
    Ipp32f silFloor, Ipp32f maxE, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_16s(Ipp16s* pSrcDst, int step, int height,
    Ipp16s silFloor, Ipp16s maxE, Ipp16s val, Ipp32f enScale);

```

Parameters

pSrcDst Pointer to the input and output vector [*height*step*].

<i>step</i>	Sample step in the vector <i>pSrcDst</i> .
<i>height</i>	Number of samples for the normalization.
<i>silFloor</i>	Silence floor value.
<i>val</i>	Coefficient value.
<i>maxE</i>	Maximum energy value.
<i>enScale</i>	Energy scale.

Description

The function `ippsNormEnergy` is declared in the `ippsr.h` file. This function normalizes the input vector that contains energy values.

The normalization is performed as follows:

For functions without the RT suffix, the maximum energy value is calculated as

$$maxE = \max_{0 < i < height - 1} pSrcDst[i \cdot step]$$

For all functions, assuming $val = 1$ if not specified,

$$minE = maxE - (silFloor \cdot \ln 10) / 10$$

$$pSrcDst[i \cdot step] = val - (maxE - \max(pSrcDst[i \cdot step], minE)) \cdot enScale,$$

for $0 \leq i < height$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.

SumMeanVar

Calculates both the sum of a the vector and its square sum.

Syntax

```
IppStatus ippsSumMeanVar_32f(const Ipp32f* pSrc, int srcStep, int height,
    Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsSumMeanVar_32f_I(const Ipp32f* pSrc, int srcStep, int height,
    Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
IppStatus ippsSumMeanVar_16s32f(const Ipp16s* pSrc, int srcStep, int height,
    Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsSumMeanVar_16s32f_I(const Ipp16s* pSrc, int srcStep, int height,
    Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
IppStatus ippsSumMeanVar_16s32s_Sfs(const Ipp16s* pSrc, int srcStep, int
    height, Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsSumMeanVar_16s32s_ISfs(const Ipp16s* pSrc, int srcStep, int
    height, Ipp32s* pSrcDstMean, Ipp32s* pSrcDstVar, int width, int
    scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the source vector [<i>height*srcStep</i>].
<i>srcStep</i>	Row step in the vector <i>pSrc</i> .
<i>height</i>	Number of rows in <i>pSrc</i>
<i>pDstMean</i>	Pointer to the destination vector that contains the sums [<i>width</i>].
<i>pDstVar</i>	Pointer to the destination vector that contains the square sums [<i>width</i>].
<i>pSrcDstMean</i>	Pointer to the source and destination vector that contains the sums [<i>width</i>].
<i>pSrcDstVar</i>	Pointer to the source and destination vector that contains the square sums [<i>width</i>].
<i>width</i>	Number of columns in the source vector <i>pSrc</i> .
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsSumMeanVar` is declared in the `ippsr.h` file. This function calculates both the sums and the square sums of the source vectors as follows:

$$pDstVar[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pDstMean[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] ,$$

for $j = 0, \dots, width-1$.

The function `ippsSumMeanVar_I` performs the in-place calculation as given by:

$$pSrcDstVar[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pSrcDstMean[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] ,$$

for $j = 0, \dots, width-1$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> , <code>pDstMean</code> , <code>pDstVar</code> , <code>pSrcDstMean</code> , or <code>pSrcDstVar</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>srcStep</code> , <code>width</code> , or <code>height</code> is less than or equal to 0, or when <code>width</code> is greater than <code>srcStep</code> .

NewVar

Calculates the variances given the sum and square sum accumulators.

Syntax

```
IppStatus ippsNewVar_32f(const Ipp32f* pSrcMean, const Ipp32f* pSrcVar, Ipp32f*
    pDstVar, int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32f_I(const Ipp32f* pSrcMean, Ipp32f* pSrcDstVar,
    int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32s_Sfs(const Ipp32s* pSrcMean, const Ipp32s* pSrcVar,
    Ipp32s* pDstVar, int width, Ipp32f val1, Ipp32f val2, int scaleFactor);
IppStatus ippsNewVar_32s_ISfs(const Ipp32s* pSrcMean, Ipp32s* pSrcDstVar,
    int width, Ipp32f val1, Ipp32f val2, int scaleFactor);
```

Parameters

<i>pSrcMean</i>	Pointer to the vector that accumulates sums [<i>width</i>].
<i>pSrcVar</i>	Pointer to the vector that accumulates square sums [<i>width</i>].
<i>pSrcDstVar</i>	Pointer to the square sum accumulators and the resulting variance vector [<i>width</i>].
<i>pDstVar</i>	Pointer to the result variance vector [<i>width</i>].
<i>width</i>	Length of the variance vector <i>pDstVar</i> .
<i>val1, val2</i>	Constant coefficients.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsNewVar` is declared in the `ippsr.h` file. This function calculates the variance values given the sum and square sum accumulators as follows:

$$pDstVar[i] = (pSrcVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2, i=0..width-1$$

whereas the in-place function `ippsNewVar_I` uses the following formula:

$$pSrcDstVar[i] = (pSrcDstVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2,$$

for $i = 0, \dots, width - 1$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcMean</i> , <i>pSrcVar</i> , <i>pDstVar</i> , or <i>pSrcDstVar</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> is less than or equal to 0.

RecSqrt

*Calculates square roots of a vector
and their reciprocals.*

Syntax

```

IppStatus ippsRecSqrt_32s_Sfs(Ipp32s* pSrcDst, int len, Ipp32s val, int
    scaleFactor);
IppStatus ippsRecSqrt_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
IppStatus ippsRecSqrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
    Ipp32s val, int scaleFactor);

```

Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector [<i>len</i>].
<i>len</i>	Length of the vector <i>pSrcDst</i> .
<i>val</i>	Threshold for processed source values.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsRecSqrt` is declared in the `ippsr.h` file. This function calculates the square root of a vector and then takes the reciprocals. The operations are as follows:

$$pSrcDst[j] = \begin{cases} val & , \text{ if } pSrcDst[j] < val \\ \frac{1}{\sqrt{pSrcDst[j]}} & , \text{ otherwise} \end{cases}$$

for $j = 0, \dots, len - 1$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0 or <code>val</code> is less than or equal to 0.
<code>ippStsInvZero</code>	Indicates a warning that all <code>pSrcDst[i]</code> are less than <code>val</code> .

AccCovarianceMatrix

Accumulates covariance matrix.

Syntax

```
IppStatus ippAccCovarianceMatrix_16s64f_D2L(const Ipp16s** mSrc, int height,
      const Ipp16s* pMean, Ipp64f** mSrcDst, int width, Ipp64f val);
IppStatus ippAccCovarianceMatrix_32f64f_D2L(const Ipp32f** mSrc, int height,
      const Ipp32f* pMean, Ipp64f** mSrcDst, int width, Ipp64f val);
IppStatus ippAccCovarianceMatrix_16s64f_D2(const Ipp16s* pSrc, int srcStep,
      int height, const Ipp16s* pMean, Ipp64f* pSrcDst, int width, int dstStep,
      Ipp64f val);
IppStatus ippAccCovarianceMatrix_32f64f_D2(const Ipp32f* pSrc, int srcStep,
      int height, const Ipp32f* pMean, Ipp64f* pSrcDst, int width, int dstStep,
      Ipp64f val);
```

Parameters

<code>pSrc</code>	Pointer to the input vector [$height * srcStep$].
<code>mSrc</code>	Pointer to the input matrix [$height$][$width$].
<code>srcStep</code>	Row step in <code>pSrc</code> .
<code>pMean</code>	Pointer to the mean vector [$width$].
<code>width</code>	Length of the input matrix row, also length of the mean and variance vectors.
<code>pSrcDst</code>	Pointer to the result vector [$width * dstStep$].

<i>mSrcDst</i>	Pointer to the result matrix [<i>width</i>][<i>width</i>].
<i>dstStep</i>	Row step in <i>pSrcDst</i> .
<i>height</i>	Number of rows in the input matrix.
<i>val</i>	Value to multiply to each distance.

Description

The function `ippsAccCovarianceMatrix` is declared in the `ippsr.h` file. This function accumulates destination covariance matrix elements according to formulas below.

For functions with the D2 suffix,

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] + \\ + val \cdot \sum_{k=0}^{height-1} (pSrc[k \cdot srcStep + i] - pMean[j]) \cdot (pSrc[k \cdot srcStep + j] - pMean[j])$$

$$pSrcDst[j \cdot dstStep + i] = pSrcDst[i \cdot dstStep + j] ,$$

for $i = 0, \dots, width-1, j = i, \dots, width-1$.

For functions with the D2L suffix,

$$mSrcDst[i][j] = mSrcDst[i][j] + val \cdot \sum_{k=0}^{height-1} (mSrc[k][i] - pMean[j]) \cdot (mSrc[k][j] - pMean[j])$$

$$mSrcDst[j][i] = mSrcDst[i][j] ,$$

for $i = 0, \dots, width-1, j = i, \dots, width-1$.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>mSrc</i> , <i>pMean</i> , <i>pSrcDst</i> , or <i>mSrcDst</i> pointer is NULL.
<code>ippsStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippsStsStrideErr</code>	Indicates an error when <i>srcStep</i> or <i>dstStep</i> is less than <i>width</i> .

Derivative Functions

Functions described in this section are used to calculate first and second derivatives of feature vectors. These functions process the sequence of input feature vectors and generate the corresponding output feature vectors that contain derivatives.

The input sequence a_0, \dots, a_{N-1} is stored as a one-dimensional array of length $N \cdot M$, where N is the number of feature vectors and M is the dimension of each feature a_i . Similarly, the output sequence b_0, \dots, b_{N-1} is also stored as a one-dimensional array of length $N \cdot K$, where K is the dimension of each feature b_j .

Certain constraints on values of M and K must be observed, specifically,

$$K \geq M;$$

$$K \geq 2M, \text{ for generating first derivatives;}$$

$$K \geq 3M, \text{ for generating both first and second derivatives.}$$

The `ippsCopyColumn` function copies the input sequence to the output sequence, that is, places the base features into the output sequence. The base features are placed in the first M elements of each output vector. The function `ippsEvalDelta` is then used to calculate

the derivatives. While these two functions are used for general derivative calculation, the functions `ippsDelta` and `ippsDeltaDelta` provide combined but specific functionalities.

As the derivative operation accesses feature vectors both in the history and in the future, special treatment is required for the first `winSize` (delta window size) features, as well as for the last `winSize` features. For the first `winSize` features, the first feature vector is usually repeated to provide the history. For the last `winSize` features, the last feature vector is repeated to provide the future information.

CopyColumn

Copies the input sequence into the output sequence.

Syntax

```
IppStatus ippsCopyColumn_16s_D2(const Ipp16s* pSrc, int srcWidth,
    Ipp16s* pDst, int dstWidth, int height);

IppStatus ippsCopyColumn_32f_D2(const Ipp32f* pSrc, int srcWidth,
    Ipp32f* pDst, int dstWidth, int height);
```

Parameters

<i>pSrc</i>	Pointer to the input feature sequence [<i>height*srcWidth</i>].
<i>srcWidth</i>	Length of each input feature vector.
<i>pDst</i>	Pointer to the output feature sequence [<i>height*dstWidth</i>].
<i>dstWidth</i>	Length of each output feature vector.
<i>height</i>	Number of features in the sequence.

Description

The function `ippsCopyColumn` is declared in the `ippsr.h` file. This function copies the input feature sequence into the output sequence as follows:

$$pDst[j \cdot dstWidth + i] = pSrc[j \cdot srcWidth + i], 0 \leq i < srcWidth, 0 \leq j < height$$

The unspecified elements in the output sequence remain unchanged.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>srcWidth</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>dstWidth</i> is less than <i>srcWidth</i> .

EvalDelta

Calculates the derivatives of feature vectors.

Syntax

```

IppStatus ippsEvalDelta_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step, int
    width, int offset, int winSize, Ipp16s val, int scaleFactor);

IppStatus ippsEvalDelta_32f_D2(Ipp32f* pSrcDst, int height, int step, int
    width, int offset, int winSize, Ipp32f val);

IppStatus ippsEvalDeltaMul_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step,
    const Ipp16s* pVal, int width, int offset, int winSize, int scaleFactor);

```

```
IppStatus ippsEvalDeltaMul_32f_D2(Ipp32f* pSrcDst, int height, int step, const
    Ipp32f* pVal, int width, int offset, int winSize);
```

Parameters

<i>pSrcDst</i>	Pointer to the input and output sequence [<i>height*step</i>].
<i>height</i>	Number of features in <i>pSrcDst</i> .
<i>step</i>	Length of each feature in <i>pSrcDst</i> .
<i>width</i>	The number of derivatives to be calculated for each feature.
<i>offset</i>	Offset to place the derivative values.
<i>winSize</i>	The delta window size
<i>val</i>	The delta coefficient.
<i>pVal</i>	Pointer to the delta coefficients vector [<i>width</i>].
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsEvalDelta` is declared in the `ippsr.h` file. This function calculates the derivatives for the input feature sequence. The base feature ranges from *offset* to (*offset* + *width* - 1) in each feature vector. The output derivatives are stored in each feature vector next to the base features ranging from (*offset* + *width*) to (*offset* + 2**width* - 1). The operations are detailed as follows:

For `ippsEvalDelta` functions,

$$pSrcDst[i \cdot step + width + j] = val \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$$0 \leq i < height, \quad offset \leq j < offset + width.$$

For `ippsEvalDeltaMul` functions,

$$pSrcDst[i \cdot step + width + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$0 \leq i < \text{height}, \quad \text{offset} \leq j < \text{offset} + \text{width}.$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>width</i> , or <i>winSize</i> is less than or equal to 0; or <i>offset</i> is less than 0; or <i>height</i> is less than $2 * \text{winSize}$.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than $\text{offset} + 2 * \text{width}$.

Delta

Copies the base features and calculates the derivatives of feature vectors.

Syntax

```

IppStatus ippDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
    pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);
IppStatus ippDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
    pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);
IppStatus ippDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f* pDst,
    int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f* pDst,
    int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
    int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
    scaleFactor);
IppStatus ippDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
    int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
    scaleFactor);
IppStatus ippDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal, int
    srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
IppStatus ippDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal, int
    srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);

```

Parameters

<i>pSrc</i>	Pointer to the input feature sequence [<i>height</i> * <i>srcWidth</i>].
<i>srcWidth</i>	Length of the feature vector in the input sequence <i>pSrc</i> .
<i>pDst</i>	Pointer to the output feature sequence
<i>dstStep</i>	Length of the feature vector in the output sequence <i>pDst</i> .
<i>height</i>	Number of feature vectors.
<i>val</i>	Delta coefficient.
<i>deltaMode</i>	Execution mode.
<i>pVal</i>	Pointer to the delta coefficients vector [<i>width</i>].
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The functions `ippsDelta` and `ippsDeltaMul` are declared in the `ippsr.h` file. These functions provide a combined functionality of `ippsCopyColumn` and `ippsEvalDelta`. First, the input feature vectors are copied to the output sequence. Then the derivatives are calculated.

In the subsequent discussion, the following conditions hold:

The function suffix `win1` or `win2` specifies the delta window size, *winSize* = 1 or 2, respectively;

The function `ippsDelta` implies the following constraints on the delta coefficients:

$$pVal[j] \equiv val, \quad \forall j.$$

The execution mode *deltaMode* provides additional controls along the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to `IPP_DELTA_BEGIN|IPP_DELTA_END`

Perform the offline delta feature calculation. All base features are assumed available at the time of the calculation. The base features are copied from the input stream *pSrc* to the output stream *pDst* as follows:

$$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcWidth + j], \quad (8.4)$$

$$0 \leq i < height, \quad 0 \leq j < srcWidth.$$

Then the derivatives are calculated as given by:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} \{pDst[\min(i+k, height-1) \cdot dstStep + j] - pDst[\max(i-k, 0) \cdot dstStep + j]\}, \quad (8.5)$$

for $0 \leq i < height$, $0 \leq j < srcWidth$.

2. *deltaMode* is equal to 0

Perform the online delta feature calculation. The input features are the current segment of a continuous stream. The function `ippsDelta` calculates the partial delta features accordingly. First, the base features are copied as follows:

$$pDst[(i+2 \cdot winSize) \cdot dstStep + j] = pSrc[i \cdot srcWidth + j], \quad (8.6)$$

for $0 \leq i < height$, $0 \leq j < srcWidth$.

Then the derivatives are calculated as given by:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} \{pDst[(i+k) \cdot dstStep + j] - pDst[(i-k) \cdot dstStep + j]\}, \quad (8.7)$$

for $0 \leq i - winSize < height$, $0 \leq j < srcWidth$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the derivatives in $pDst[k \cdot dstStep + srcWidth + j]$ are assumed available through a previous derivative calculation for $0 \leq j < srcWidth$, $0 \leq i < 2 \cdot winSize$, and $0 \leq k < winSize$.

3. *deltaMode* is equal to `IPP_DELTA_BEGIN`

Perform the partial online delta feature calculation, where the beginning of the input stream is known. First, the base features are copied as in equation (8.4). Then, the derivatives are calculated as follows:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} \{pDst[(i+k) \cdot dstStep + j] - pDst[\max(i-k, 0) \cdot dstStep + j]\}, \quad (8.8)$$

for $0 \leq i < height - winSize$, $0 \leq j < srcWidth$.

4. *deltaMode* is equal to `IPP_DELTA_END`

Perform the partial online delta feature calculation, where the ending of the input stream is known. First, the base features are copied as in equation (8.6). Then, the derivatives are calculated as follows:

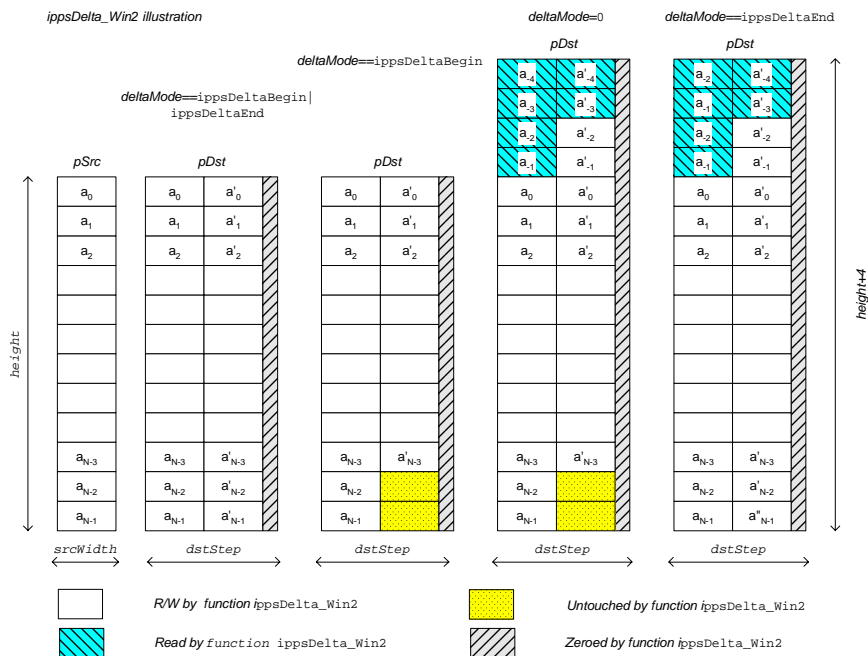
$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} \{ pDst[\min(i+k, height + 2 \cdot winSize - 1) \cdot dstStep + j] - pDst[(i-k) \cdot dstStep + j] \}$$

for $0 \leq i - winSize < height + winSize$, $0 \leq j < srcWidth$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the derivatives in $pDst[k \cdot dstStep + srcWidth + j]$ are assumed available through a previous derivative calculation for $0 \leq j < srcWidth$, $0 \leq i < 2 \cdot winSize$, and $0 \leq k < winSize$.

The following figure illustrates the above four delta calculation modes:

Figure 8-1 Execution Modes of ippsDelta_win2 function



Return Value

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pDst</i> , or <i>pVal</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>srcWidth</i> is less than or equal to 0; or <i>height</i> is less than 0; or <i>height</i> is less than $2*winSize$ when <i>deltaMode</i> is equal to <code>IPP_DELTA_BEGIN</code> ; or <i>height</i> is equal to 0 when <i>deltaMode</i> is not equal to <code>IPP_DELTA_END</code> .
<code>ippStsStrideErr</code>	Indicates an error when <i>dstStep</i> is less than $2*srcWidth$.

DeltaDelta

Copies the base features and calculates their first and second derivatives.

Syntax

```
IppStatus ippsDeltaDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
    Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
    deltaMode, int scaleFactor);

IppStatus ippsDeltaDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
    Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
    deltaMode, int scaleFactor);

IppStatus ippsDeltaDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
    pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);

IppStatus ippsDeltaDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
    pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);

IppStatus ippsDeltaDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
    pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
    int scaleFactor);

IppStatus ippsDeltaDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
    pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
    int scaleFactor);

IppStatus ippsDeltaDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal,
    int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);

IppStatus ippsDeltaDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal,
    int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Parameters

<i>pSrc</i>	Pointer to the input feature sequence [<i>height</i> * <i>srcWidth</i>].
<i>srcWidth</i>	Length of the input feature in the input sequence <i>pSrc</i> .
<i>pDst</i>	Pointer to the output feature sequence.
<i>dstStep</i>	Length of the output feature in the output sequence <i>pDst</i> .
<i>height</i>	Number of feature vectors.
<i>val1</i> , <i>val2</i>	The first and second delta coefficients.

<i>deltaMode</i>	Execution mode.
<i>pVal</i>	Pointer to the delta coefficients vector [<i>width</i>].
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The functions `ippsDeltaDelta` and `ippsDeltaDeltaMul` are declared in the `ippsr.h` file. These functions provide a combined functionality of `ippsCopyColumn` and double operations of `ippsEvalDelta`. First, the input feature vectors are copied to the output sequence. Then the first and second derivatives are calculated.

In the subsequent discussion, the following conditions hold:

The function suffix `win1` or `win2` specifies the delta window size, `winSize = 1` or `2`, respectively;

The function `ippsDeltaDelta` implies the following constraints on the delta coefficients:

$pVal[j] \equiv val1, \forall j$, for the first derivative calculation and
 $pVal[j] \equiv val2, \forall j$, for the second derivative calculation.

The execution mode *deltaMode* provides additional controls along the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to `IPP_DELTA_BEGIN|IPP_DELTA_END`
Perform the offline delta-delta feature calculation. All base features are assumed available at the time of the calculation. First, the base features are copied from the input stream *pSrc* to the output stream *pDst* according to (8.4). Then the first derivatives are calculated as given by (8.5) for $0 \leq i < height$, $0 \leq j < srcWidth$. Finally, the second derivatives are calculated also by formula (8.5) for $0 \leq i < height$, $0 \leq j - srcWidth < srcWidth$.
2. *deltaMode* is equal to `0`
Perform the online delta-delta feature calculation. The input features are the current segment of a continuous stream. The function `ippsDeltaDelta` calculates the partial delta-delta features accordingly.

First, the base features are copied as follows:

$$pDst[(i+3 \cdot winSize) \cdot dstStep+j] = pSrc[i \cdot srcWidth+j] , \quad (8.9)$$

for $0 \leq i < height$, $0 \leq j < srcWidth$.

Then the first derivatives are calculated as given by (8.7), for $0 \leq i-2 \cdot winSize < height$ and $0 \leq j < srcWidth$.

Finally, the second derivatives are calculated also by formula (8.7) for $0 \leq i-winSize < height$, $0 \leq j-srcWidth < srcWidth$.

In this execution mode, the base features in $pDst[i \cdot dstStep+j]$, the first derivatives in $pDst[k \cdot dstStep+srcWidth+j]$, and the second derivatives in $pDst[l \cdot dstStep+2 \cdot srcWidth+j]$ are assumed available through a previous delta-delta calculation for $0 \leq j < srcWidth$, $0 \leq i < 3 \cdot winSize$, $0 \leq k < 2 \cdot winSize$, and $0 \leq l < winSize$.

3. *deltaMode* is equal to IPP_DELTA_BEGIN

Perform the partial online delta-delta feature calculation, where the beginning of the input stream is known. Initially, the base features are copied as in equation (8.4).

Then, the first derivatives are calculated as given by (8.8), for $0 \leq i < height-winSize$ and $0 \leq j < srcWidth$.

Finally, the second derivatives are calculated also by formula (8.8) for $0 \leq i < height-2 \cdot winSize$ and $0 \leq j-srcWidth < srcWidth$.

4. *deltaMode* is equal to IPP_DELTA_END

Perform the partial online delta-delta feature calculation, where the ending of the input stream is known. Initially, the base features are copied as in equation (8.9).

Then, the first derivatives are calculated as follows:

$$pDst[i \cdot dstStep+srcWidth+j] = pVal[j] \cdot \sum_{k=1}^{winSize} \{ \quad (8.10)$$

$$pDst[\min(i+k, height+3 \cdot winSize-1) \cdot dstStep+j] - pDst[(i-k) \cdot dstStep+j] \}$$

for $0 \leq i-2 \cdot winSize < height+winSize$, $0 \leq j < srcWidth$.

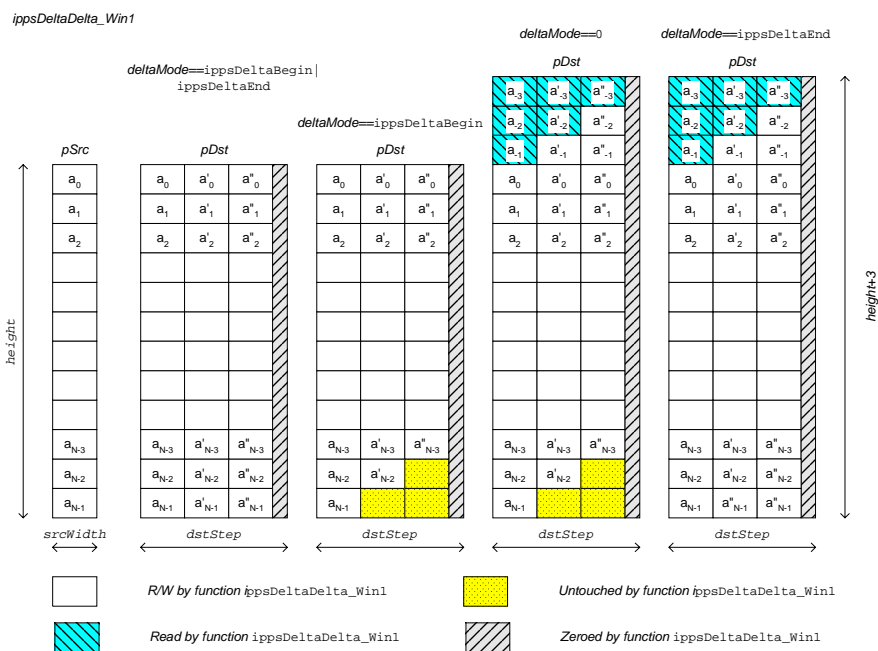
Finally, the second derivatives are calculated also according to (8.10) where the indexes i and j vary in the following ranges:

$$0 \leq i - \text{winSize} < \text{height} + 2 \cdot \text{winSize}, 0 \leq j - \text{srcWidth} < \text{srcWidth}.$$

In this execution mode, the base features in $pDst[i \cdot \text{dstStep} + j]$, the first derivatives in $pDst[k \cdot \text{dstStep} + \text{srcWidth} + j]$, and the second derivatives in $pDst[l \cdot \text{dstStep} + 2 \cdot \text{srcWidth} + j]$ are assumed available through a previous delta-delta calculation for $0 \leq j < \text{srcWidth}$, $0 \leq k < 3 \cdot \text{winSize}$, $0 \leq l < 2 \cdot \text{winSize}$, and $0 \leq i < \text{winSize}$.

The following figure illustrates the above four delta-delta calculation modes:

Figure 8-2 Execution Modes of `ippsDeltaDelta_win1` function



Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when `pSrc`, `pDst` or `pVal` pointer is NULL.

<code>ippStsSizeErr</code>	Indicates an error when <i>srcWidth</i> is less than or equal to 0; or <i>height</i> is less than 0; or <i>height</i> is less than $3 * \text{winSize}$ when <i>deltaMode</i> is equal to <code>IPP_DELTA_BEGIN</code> ; or <i>height</i> is equal to 0 when <i>deltaMode</i> is not equal to <code>IPP_DELTA_END</code> .
<code>ippStsStrideErr</code>	Indicates an error when <i>dstStep</i> is less than $3 * \text{srcWidth}$.

Pitch Super Resolution

This section describes functions that can be used for implementing the pitch resolution algorithm (see [Med91]).

Some of these functions are used in the Intel IPP Speech Recognition Sample. See *Speech Processing* sample downloadable from
<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

CrossCorrCoeffDecim

Calculates vector of cross correlation coefficients with decimation.

Syntax

```
IppStatus ippsCrossCorrCoeffDecim_16s32f(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int maxLen, int minLen, Ipp32f* pDst, int dec);
```

Parameters

<i>pSrc1</i>	Pointer to the first input vector [<i>maxLen</i>].
<i>pSrc2</i>	Pointer to the second input vector [<i>maxLen</i>].
<i>pDst</i>	Pointer to the output vector [$(\text{maxLen} - \text{minLen}) / \text{dec} + 1$].
<i>maxLen</i>	Maximal length of cross correlation.
<i>minLen</i>	Minimal length of cross correlation.
<i>dec</i>	Decimation step.

Description

The function `ippsCrossCorrCoeffDecim` is declared in the `ippsr.h` file. This function calculates the vector of cross correlation coefficients with lengths from *minLen* to *maxLen* using the decimation step *dec*.

Computations are performed as given by:

$$pDst[k] = \begin{cases} 0, & \text{if } (\mathbf{x}^k, \mathbf{x}^k)_{dec} \cdot (\mathbf{y}^k, \mathbf{y}^k)_{dec} = 0 \\ \frac{(\mathbf{x}^k, \mathbf{y}^k)_{dec}}{\sqrt{(\mathbf{x}^k, \mathbf{x}^k)_{dec}} \cdot \sqrt{(\mathbf{y}^k, \mathbf{y}^k)_{dec}}}, & \text{otherwise} \end{cases}$$

where

$$\mathbf{x}_i^k = pSrc1[maxLen - minLen - dec \cdot k + i], \quad \mathbf{y}_i^k = pSrc2[i],$$

$$i = 0, \dots, minLen + dec \cdot k - 1, \quad k = 0, \dots, (maxLen - minLen) / dec,$$

and $(\mathbf{a}, \mathbf{b})_{dec}$ denotes the decimated dot product of two vectors \mathbf{a} and \mathbf{b} :

$$(\mathbf{a}, \mathbf{b})_{dec} = \sum_{i=0}^{(len-1)/dec} a[i \cdot dec] \times b[i \cdot dec]$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>minLen</i> or <i>dec</i> is less than or equal to 0, or if <i>maxLen</i> is less than <i>minLen</i> .

CrossCorrCoeff

Calculates the cross correlation coefficient.

Syntax

```
IppStatus ippsCrossCorrCoeff_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp32f* pResult);
```

```
IppStatus ippsCrossCorrCoeffPartial_16s32f(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32f val, Ipp32f* pResult);
```

Parameters

<i>pSrc1</i>	Pointer to the first input vector [<i>len</i>].
<i>pSrc2</i>	Pointer to the second input vector [<i>len</i>].
<i>len</i>	Length of the input vectors.
<i>pResult</i>	Pointer to the result cross correlation coefficient value.
<i>val</i>	Value used as the squared magnitude of the first vector.

Description

The functions `ippsCrossCorrCoeff` and `ippsCrossCorrCoeffPartial` are declared in the `ippsr.h` file. These functions calculate the cross correlation coefficient of two vectors according to the formulas given below.

For the `ippsCrossCorrCoeff` function:

$$pResult[0] = \begin{cases} 0, & \text{if } (\mathbf{x}, \mathbf{x}) \cdot (\mathbf{y}, \mathbf{y}) = 0 \\ \frac{(\mathbf{x}, \mathbf{y})}{\sqrt{(\mathbf{x}, \mathbf{x})} \cdot \sqrt{(\mathbf{y}, \mathbf{y})}}, & \text{otherwise} \end{cases}$$

For the `ippsCrossCorrCoeffPartial` function:

$$pResult[0] = \begin{cases} 0, & \text{if } val \cdot (\mathbf{y}, \mathbf{y}) = 0 \\ \frac{(\mathbf{x}, \mathbf{y})}{\sqrt{val} \cdot \sqrt{(\mathbf{y}, \mathbf{y})}}, & \text{otherwise} \end{cases}$$

where

$\mathbf{x}_i = pSrc1[i-1]$, $\mathbf{y}_i = pSrc2[i-1]$, for $i = 1, \dots, len$,

and (\mathbf{a}, \mathbf{b}) denotes the dot product of two vectors \mathbf{a} and \mathbf{b} .

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , or <i>pResult</i> pointer is NULL.

<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates an error when <code>val</code> is less than 0.

CrossCorrCoeffInterpolation

Calculates interpolated cross correlation coefficient.

Syntax

```
ippStatus ippsCrossCorrCoeffInterpolation_16s32f(const Ipp16s* pSrc1, const
Ipp16s* pSrc2, int len, Ipp32f* pBeta, Ipp32f* pResult);
```

Parameters

<code>pSrc1</code>	Pointer to the first input vector [<code>len</code>].
<code>pSrc2</code>	Pointer to the second input vector [<code>len+1</code>].
<code>len</code>	Length of the input vectors.
<code>pResult</code>	Pointer to the result cross correlation coefficient value.
<code>pBeta</code>	Pointer to the result value of fractional part of the pitch period.

Description

The function `ippsCrossCorrCoeffInterpolation` is declared in the `ippsr.h` file. This function calculates interpolated cross correlation coefficient `pResult` of two vectors and the fractional part `pBeta` of the pitch period according to the following formulas:

$$pBeta[0] = \beta = \frac{(\mathbf{x}, \mathbf{y}') \cdot (\mathbf{y}, \mathbf{y}) - (\mathbf{x}, \mathbf{y}) \cdot (\mathbf{y}, \mathbf{y}')}{(\mathbf{x}, \mathbf{y}') \cdot [(\mathbf{y}, \mathbf{y}) - (\mathbf{y}, \mathbf{y}')] + (\mathbf{x}, \mathbf{y}) \cdot [(\mathbf{y}', \mathbf{y}') - (\mathbf{y}, \mathbf{y}')]} ,$$

and if $0 \leq \beta < 1$, then

$$pResult[0] = \frac{(1 - \beta) \cdot (\mathbf{x}, \mathbf{y}) + \beta \cdot (\mathbf{x}, \mathbf{y}')}{\sqrt{(\mathbf{x}, \mathbf{x}) \cdot ((1 - \beta)^2 \cdot (\mathbf{y}, \mathbf{y}) + 2\beta \cdot (1 - \beta) \cdot (\mathbf{y}, \mathbf{y}') + \beta^2 \cdot (\mathbf{y}', \mathbf{y}'))}} ,$$

where

$\mathbf{x}_i = pSrc1[i-1]$, $\mathbf{y}_i = pSrc2[i-1]$, $\mathbf{y}'_i = pSrc2[i]$, $i = 1, \dots, len$,

and (\mathbf{a}, \mathbf{b}) denotes the dot product of two vectors \mathbf{a} and \mathbf{b} .

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , <i>pBeta</i> or <i>pResult</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>srcStep</i> or <i>dstStep</i> is less than <i>width</i> .
<code>ippStsDivByZero</code>	Indicates a warning that the denominator in formula for <i>pBeta</i> [0] has zero value. Operation execution is not aborted.

Model Evaluation

This section describes functions that evaluate the acoustic and language models.

Some of these functions are used in the Intel IPP Speech Recognition Samples. See *Gaussian Calculations* sample downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

AddNRows

Adds N vectors from a vector array.

Syntax

```
IppStatus ippsAddNRows_32f_D2(Ipp32f* pSrc, int height, int offset,
    int step, Ipp32s* pInd, Ipp16u* pAddInd, int rows, Ipp32f* pDst,
    int width, Ipp32f weight);
```

Parameters

<i>pSrc</i>	Pointer to the vector array [<i>height*step</i>].
<i>height</i>	Number of rows in the vector array <i>pSrc</i> .
<i>offset</i>	Offset to the vector of interest in the vector array <i>pSrc</i> .
<i>step</i>	Row step in the vector array <i>pSrc</i> .
<i>pInd</i>	Pointer to the index vector [<i>rows</i>].
<i>pAddInd</i>	Pointer to the additional index vector [<i>rows</i>].
<i>rows</i>	Number of vectors to be added.
<i>pDst</i>	Pointer to the output vector [<i>width</i>].
<i>width</i>	Length of the output vector <i>pDst</i> .
<i>weight</i>	Weight value to be added to the output.

Description

The function `ippsAddNRows` is declared in the `ippsr.h` file. This function calculates the sum of the `rows` number of vectors, according to the sum of the indexing vectors `pInd` and `pAddInd`, from the vector array `pSrc` as follows:

$$pDst[k] = weight + \sum_{i=0}^{rows} pSrc[k + offset + step \cdot (pInd[i] + pAddInd[i])],$$

$$0 \leq k < width.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> , <code>pInd</code> , <code>pAddInd</code> , or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>height</code> , <code>width</code> , or <code>rows</code> is less than or equal to 0; or $(pInd[i] + pAddInd[i])$ or <code>offset</code> is less than 0; or $(pInd[i] + pAddInd[i])$ is greater than or equal to <code>height</code> .
<code>ippStsStrideErr</code>	Indicates an error when <code>step</code> is less than <code>width + offset</code> .

ScaleLM

Scales vector elements with thresholding.

Syntax

```
IppStatus ippsScaleLM_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len, Ipp16s
    floor, Ipp16s scale, Ipp32s base);
IppStatus ippsScaleLM_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
    floor, Ipp32f scale, Ipp32f base);
```

Parameters

<code>pSrc</code>	Pointer to the input vector [<code>len</code>].
<code>pDst</code>	Pointer to the output vector [<code>len</code>].
<code>len</code>	Length of the vector <code>pSrc</code> or <code>pDst</code> .

<i>floor</i>	Threshold value.
<i>scale</i>	Scaling factor.
<i>base</i>	Additive factor.

Description

The function `ippsScaleLM` is declared in the `ippsr.h` file. This function sets threshold on the input vector *pSrc* and scales the vector elements as follows:

$$pDst[n] = scale \cdot \max(pSrc[n], floor) + base, \quad 0 \leq n < len$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

LogAdd

Adds two vectors in the logarithmic representation.

Syntax

```
IppStatus ippsLogAdd_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len,
    IppHintAlgorithm hint);
IppStatus ippsLogAdd_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
    IppHintAlgorithm hint);
IppStatus ippsLogAdd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    scaleFactor, IppHintAlgorithm hint);
IppStatus ippsLogAdd_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    scaleFactor, IppHintAlgorithm hint);
```

Parameters

<i>pSrc</i>	Pointer to the first input vector [<i>len</i>].
<i>pSrcDst</i>	Pointer to the second input vector and also the output vector [<i>len</i>].
<i>len</i>	Number of elements in the input and output vectors.

<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.
<i>hint</i>	Suggestion for using specific code for logarithmic addition.

Description

The function `ippsLogAdd` is declared in the `ippsr.h` file. This function adds the *pSrc* and *pSrcDst* vectors, whose elements are in the logarithmic representation.

For functions with floating point arguments, the output vector *pSrcDst*, also in the logarithmic representation, is calculated as follows:

$$pSrcDst[i] = \ln(e^{pSrc[i]} + e^{pSrcDst[i]}), \quad 0 \leq i < len.$$

For functions with integer arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = 2^{scaleFactor} \cdot \ln\left(e^{pSrc[i] \cdot 2^{-scaleFactor}} + e^{pSrcDst[i] \cdot 2^{-scaleFactor}}\right),$$

$$0 \leq i < len.$$

The *hint* argument suggests using special code which provides for faster but less accurate calculation, or more accurate but slower calculation. The values you can enter for the *hint* argument are listed in [Table 7-3, “Flag and Hint Arguments”](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

LogSub

*Subtracts a vector from another vector,
in the logarithmic representation.*

Syntax

```
IppStatus ippsLogSub_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsLogSub_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsLogSub_16s_sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
    int scaleFactor);
```



```
IppStatus ippsLogSub_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len,
    int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the first input vector [<i>len</i>].
<i>pSrcDst</i>	Pointer to the second input and the output vector [<i>len</i>].
<i>len</i>	Number of elements in the input and output vectors.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Discussion

The function `ippsLogSub` is declared in the `ippsr.h` file. This function subtracts the vector *pSrcDst* from the vector *pSrc*, both of them in the logarithmic representation. For functions with floating point arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = \ln(e^{pSrc[i]} - e^{pSrcDst[i]}), \quad 0 \leq i < len$$

For functions with integer arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = 2^{scaleFactor} \cdot \ln\left(e^{pSrc[i] \cdot 2^{-scaleFactor}} - e^{pSrcDst[i] \cdot 2^{-scaleFactor}}\right),$$

$$0 \leq i < len.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsRangeErr</code>	Indicates an error when <i>pSrc[i]</i> is less than or equal to <i>pSrcDst[i]</i> .

LogSum

Sums vector elements in the logarithmic representation.

Syntax

```
IppStatus ippsLogSum_32f(const Ipp32f* pSrc, Ipp32f* pResult, int len,
    IppHintAlgorithm hint);

IppStatus ippsLogSum_64f(const Ipp64f* pSrc, Ipp64f* pResult, int len,
    IppHintAlgorithm hint);

IppStatus ippsLogSum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pResult, int len,
    int scaleFactor, IppHintAlgorithm hint);

IppStatus ippsLogSum_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pResult, int len,
    int scaleFactor, IppHintAlgorithm hint);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len</i>].
<i>pResult</i>	Pointer to the result value.
<i>len</i>	Number of elements in the input vector.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.
<i>hint</i>	Suggestion for using specific code for logarithmic addition.

Description

The function `ippsLogSum` is declared in the `ippsr.h` file. This function sums the source vector elements that are taken in logarithmic representation. For functions with floating point arguments, the output value, also in logarithmic representation, is calculated as follows:

$$pResult[0] = \ln \sum_{i=0}^{len-1} e^{pSrc[i]}$$

For functions with integer arguments, the output value is calculated as follows:

$$pResult[0] = 2^{scaleFactor} \cdot \ln \left(\sum_{i=0}^{len-1} e^{pSrc[i] \cdot 2^{-scaleFactor}} \right)$$

The *hint* argument suggests using special code which provides for faster but less accurate calculation, or more accurate but slower calculation. The values you can enter for the *hint* argument are listed in [Table 7-3, “Hint Arguments for Fourier Transform Functions”](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pResult</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

MahDistSingle

Calculates the Mahalanobis distance for a single observation vector.

Syntax

```

IppStatus ippsMahDistSingle_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s* pMean,
    const Ipp16s* pVar, int len, Ipp32s* pResult, int scaleFactor);
IppStatus ippsMahDistSingle_16s32f(const Ipp16s* pSrc, const Ipp16s* pMean,
    const Ipp16s* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_32f(const Ipp32f* pSrc, const Ipp32f* pMean, const
    Ipp32f* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_64f(const Ipp64f* pSrc, const Ipp64f* pMean, const
    Ipp64f* pVar, int len, Ipp64f* pResult);
IppStatus ippsMahDistSingle_32f64f(const Ipp32f* pSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int len, Ipp64f* pResult);

```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len</i>].
<i>pMean</i>	Pointer to the mean vector [<i>len</i>].
<i>pVar</i>	Pointer to the variance vector [<i>len</i>].
<i>len</i>	Number of elements in the input, mean, and variance vectors.
<i>pResult</i>	Pointer to the result.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsMahDistSingle` is declared in the `ippsr.h` file. This function calculates the Mahalanobis distance for the input vector `pSrc`, given the mean vector `pMean` and the variance vector `pVar`. The calculation is as follows:

$$pResult[0] = \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> , <code>pMean</code> , <code>pVar</code> , or <code>pResult</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

MahDist

*Calculates the Mahalanobis distances
for multiple observation vectors.*

Syntax

```
IppStatus ippsMahDist_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f* pMean,
    const Ipp32f* pVar, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDist_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
    Ipp32f* pVar, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDist_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f* pMean,
    const Ipp64f* pVar, int width, Ipp64f* pDst, int height);
IppStatus ippsMahDist_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean, const
    Ipp64f* pVar, int width, Ipp64f* pDst, int height);
```

Parameters

<code>pSrc</code>	Pointer to the input vector [<code>height*step</code>].
<code>mSrc</code>	Pointer to the input matrix [<code>height</code>][<code>width</code>].
<code>step</code>	Row step in the input vector <code>pSrc</code> .
<code>pMean</code>	Pointer to the mean vector [<code>width</code>].

<i>pVar</i>	Pointer to the variance vector [<i>width</i>].
<i>width</i>	Length of the mean and variance vectors.
<i>pDst</i>	Pointer to the result vector [<i>height</i>].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .

Description

The function `ippsMahDist` is declared in the `ippsr.h` file. This function calculates the Mahalanobis distances for multiple input vectors, given the mean vector *pMean* and the variance vector *pVar*. The calculation is as follows:

For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, 0 \leq i < height.$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, 0 \leq i < height.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

MahDistMultiMix

*Calculates the Mahalanobis distances
for multiple means and variances.*

Syntax

```
IppStatus ippsMahDistMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f* pVar,
    int step, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
```

```

IppStatus ippsMahDistMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
    mVar, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDistMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f* pVar,
    int step, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);
IppStatus ippsMahDistMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
    mVar, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);

```

Parameters

<i>pMean</i>	Pointer to the mean vector [<i>height</i> * <i>step</i>].
<i>pVar</i>	Pointer to the variance vector [<i>height</i> * <i>step</i>].
<i>mMean</i>	Pointer to the mean matrix [<i>height</i>][<i>width</i>].
<i>mVar</i>	Pointer to the variance matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in mean and variance vectors.
<i>pSrc</i>	Pointer to the input vector [<i>width</i>].
<i>width</i>	Length of the input vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the result vector [<i>height</i>].
<i>height</i>	Length of the result vector <i>pDst</i> .

Description

The function `ippsMahDistMultiMix` is declared in the `ippsr.h` file. This function calculates the Mahalanobis distances for a single observation vector but multiple mean and variance pairs as follows:

For functions with the `D2` suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2, 0 \leq i < height.$$

For functions with the `D2L` suffix,

$$pDst[i] = \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2, 0 \leq i < height.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mMean</i> , <i>mVar</i> , <i>pMean</i> , <i>pVar</i> , or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

LogGaussSingle

Calculates the observation probability for a single Gaussian with an observation vector.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```

IppStatus ippsLogGaussSingle_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val,
    int scaleFactor);

IppStatus ippsLogGaussSingle_Scaled_16s32f(const Ipp16s* pSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val,
    int scaleFactor);

IppStatus ippsLogGaussSingle_32f(const Ipp32f* pSrc, const Ipp32f* pMean, const
    Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f val);

IppStatus ippsLogGaussSingle_32f64f(const Ipp32f* pSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f val);

IppStatus ippsLogGaussSingle_64f(const Ipp64f* pSrc, const Ipp64f* pMean, const
    Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f val);

IppStatus ippsLogGaussSingle_Low_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val, int
    scaleFactor);

IppStatus ippsLogGaussSingle_LowScaled_16s32f(const Ipp16s* pSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val, int
    scaleFactor);

```

Case 2: Operation for the diagonal covariance matrix

```

IppStatus ippsLogGaussSingle_DirectVar_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val,
    int scaleFactor);

IppStatus ippsLogGaussSingle_DirectVarScaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val,
    int scaleFactor);

IppStatus ippsLogGaussSingle_DirectVar_32f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f val);

IppStatus ippsLogGaussSingle_DirectVar_32f64f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f val);

IppStatus ippsLogGaussSingle_DirectVar_64f(const Ipp64f* pSrc, const Ipp64f*
    pMean, const Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f val);

```

Case 3: Operation for the identity covariance matrix

```

IppStatus ippsLogGaussSingle_IdVar_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
    pMean, int len, Ipp32s* pResult, Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussSingle_IdVarScaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, int len, Ipp32f* pResult, Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussSingle_IdVar_32f(const Ipp32f* pSrc, const Ipp32f* pMean,
    int len, Ipp32f* pResult, Ipp32f val);

IppStatus ippsLogGaussSingle_IdVar_32f64f(const Ipp32f* pSrc, const Ipp32f*
    pMean, int len, Ipp64f* pResult, Ipp64f val);

IppStatus ippsLogGaussSingle_IdVar_64f(const Ipp64f* pSrc, const Ipp64f* pMean,
    int len, Ipp64f* pResult, Ipp64f val);

IppStatus ippsLogGaussSingle_IdVarLow_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, int len, Ipp32s* pResult, Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussSingle_IdVarLowScaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, int len, Ipp32f* pResult, Ipp32f val, int scaleFactor);

```

Case 4: Operation for the block diagonal covariance matrix

```

IppStatus ippsLogGaussSingle_BlockDVar_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, const IppsBlockDMatrix_16s* pBlockVar, int len, Ipp32s*
    pResult, Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussSingle_BlockDVarScaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, const IppsBlockDMatrix_16s* pBlockVar, int len, Ipp32f*
    pResult, Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussSingle_BlockDVar_32f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const IppsBlockDMatrix_32f* pBlockVar, int len, Ipp32f* pResult,
    Ipp32f val);

```



```
IppStatus ippsLogGaussSingle_BlockDVar_32f64f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const IppsBlockDMatrix_32f* pBlockVar, int len, Ipp64f* pResult,
    Ipp64f val);

IppStatus ippsLogGaussSingle_BlockDVar_64f(const Ipp64f* pSrc, const Ipp64f*
    pMean, const IppsBlockDMatrix_64f* pBlockVar, int len, Ipp64f* pResult,
    Ipp64f val);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len</i>].
<i>pMean</i>	Pointer to the mean vector [<i>len</i>].
<i>pVar</i>	Pointer to the variance vector [<i>len</i>].
<i>pBlockVar</i>	Pointer to the block diagonal variance matrix.
<i>len</i>	Number of elements in the input, mean, and variance vectors.
<i>pResult</i>	Pointer to the result.
<i>val</i>	Gaussian constant.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussSingle` is declared in the `ippsr.h` file. This function calculates the observation probability for a single observation vector and a Gaussian mixture component. The result *pResult* is in the logarithmic representation.

For functions without the `DirectVar`, `IdVar`, or `BlockDVar` suffix, the covariance matrix is assumed to be inverse diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

For functions with the `DirectVar` suffix, the covariance matrix is assumed to be diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2 / pVar[i]$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2$$

For functions with the `BlockDVar` suffix, the covariance matrix is assumed to be block diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} \sum_{j=0}^{len-1} (pSrc[i] - pMean[i]) \cdot V[i][j] \cdot (pSrc[j] - pMean[j])$$

where $V[i][j]$ is the element of the block diagonal matrix $pVar$.

Function flavors that perform integer scaling (distinguished by the presence of `scaleFactor` argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pMean</code> , <code>pVar</code> , <code>pBlockVar</code> , or <code>pResult</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value (for <code>DirectVar</code> functions only).

LogGauss

Calculates the observation probability for a single Gaussian with multiple observation vectors.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippLogGauss_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val,
    int scaleFactor);
```

```

IppStatus ippsLogGauss_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s* pMean,
    const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val, int
    scaleFactor);

IppStatus ippsLogGauss_Scaled_16s32f_D2(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
    Ipp32f val, int scaleFactor);

IppStatus ippsLogGauss_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, Ipp32f val,
    int scaleFactor);

IppStatus ippsLogGauss_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
    pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
    Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
    pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int height, Ipp64f val);

IppStatus ippsLogGauss_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean, const
    Ipp64f* pVar, int width, Ipp64f* pDst, int height, Ipp64f val);

IppStatus ippsLogGauss_Low_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height,
    Ipp32s val, int scaleFactor);

IppStatus ippsLogGauss_Low_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val,
    int scaleFactor);

IppStatus ippsLogGauss_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
    Ipp32f val, int scaleFactor);

IppStatus ippsLogGauss_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, Ipp32f val,
    int scaleFactor);

```

Case 2: Operation for the identity covariance matrix

```

IppStatus ippsLogGauss_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
    scaleFactor);

IppStatus ippsLogGauss_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
    pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int scaleFactor);

IppStatus ippsLogGauss_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val,
    int scaleFactor);

```

```

IppStatus ippsLogGauss_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val,
    int scaleFactor);

IppStatus ippsLogGauss_IdVar_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
    pMean, int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
    int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_IdVar_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
    pMean, int width, Ipp64f* pDst, int height, Ipp64f val);

IppStatus ippsLogGauss_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
    int width, Ipp64f* pDst, int height, Ipp64f val);

IppStatus ippsLogGauss_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
    scaleFactor);

IppStatus ippsLogGauss_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
    scaleFactor);

IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int
    scaleFactor);

IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int
    scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height</i> * <i>step</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pMean</i>	Pointer to the mean vector [<i>width</i>].
<i>pVar</i>	Pointer to the variance vector [<i>width</i>].
<i>width</i>	Length of the mean and variance vectors.
<i>pDst</i>	Pointer to the result vector [<i>height</i>].
<i>height</i>	Number of rows in the input matrix, also the length of the result vector <i>pDst</i> .
<i>val</i>	Gaussian constant.

scaleFactor Scaling factor for intermediate sums.

Description

The function `ippsLogGauss` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple observation vectors for a Gaussian mixture component. The result *pResult* is in the logarithmic representation.

For functions without the `IdVar` suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional `D2` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, 0 \leq i < height$$

For functions with the additional `D2L` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, 0 \leq i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional `D2` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2, 0 \leq i < height$$

For functions with the additional `D2L` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2, 0 \leq i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>mSrc</i> , <i>pMean</i> , <i>pVar</i> , or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

LogGaussMultiMix

Calculates the observation probability for multiple Gaussian mixture components.

Syntax

```

IppStatus ippsLogGaussMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const Ipp16s*
    pVar, int step, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
    int scaleFactor);

IppStatus ippsLogGaussMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
    int scaleFactor);

IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst,
    int height, int scaleFactor);

IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst, int height,
    int scaleFactor);

IppStatus ippsLogGaussMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f* pVar,
    int step, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
    mVar, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f* pVar,
    int step, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
    mVar, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);

```

```

IppStatus ippsLogGaussMultiMix_Low_16s32s_D2Sfs(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int
    height, int scaleFactor);

IppStatus ippsLogGaussMultiMix_Low_16s32s_D2LSfs(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
    int scaleFactor);

IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int
    height, int scaleFactor);

IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, int
    scaleFactor);

```

Parameters

<i>pMean</i>	Pointer to the mean vector [<i>height</i> * <i>step</i>].
<i>pVar</i>	Pointer to the variance vector [<i>height</i> * <i>step</i>].
<i>mMean</i>	Pointer to the mean matrix [<i>height</i>][<i>width</i>].
<i>mVar</i>	Pointer to the variance matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in the mean and variance vectors.
<i>pSrc</i>	Pointer to the input vector [<i>width</i>].
<i>pSrcDst</i>	Pointer to the input and destination vector [<i>height</i>].
<i>width</i>	Number of columns in the mean matrix <i>mMean</i> .
<i>height</i>	Number of rows in the mean matrix <i>mMean</i> .
<i>scaleFactor</i>	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussMultiMix` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple Gaussian mixture components. The results are in the logarithmic representation.

The calculations are as follows:

For functions with the `D2` suffix,

$$pSrcDst[i] = pSrcDst[i] - 0.5 \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2$$

$0 \leq i < \text{height}$.

For functions with the `D2L` suffix,

$$pSrcDst[i] = pSrcDst[i] - 0.5 \sum_{j=0}^{\text{width}-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2, \quad 0 \leq i < \text{height}.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-\text{scaleFactor}}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>mMean</i> , <i>mVar</i> , <i>pMean</i> , <i>pVar</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

LogGaussMax

Calculates the likelihood probability given multiple observations and a Gaussian mixture component, using the maximum operation.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippLogGaussMax_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
    Ipp32s val, int scaleFactor);
```



```

IppStatus ippsLogGaussMax_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
    Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussMax_Scaled_16s32f_D2(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
    Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussMax_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussMax_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
    pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val);

IppStatus ippsLogGaussMax_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussMax_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
    pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height,
    Ipp64f val);

IppStatus ippsLogGaussMax_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
    const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f val);

IppStatus ippsLogGaussMax_Low_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
    Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussMax_Low_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height, Ipp32s
    val, int scaleFactor);

IppStatus ippsLogGaussMax_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
    height, Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussMax_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);

```

Case 2: Operation for the identity covariance matrix

```

IppStatus ippsLogGaussMax_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height, Ipp32s val,
    int scaleFactor);

IppStatus ippsLogGaussMax_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height, Ipp32s val,
    int scaleFactor);

```

```

IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val,
    int scaleFactor);

IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
    scaleFactor);

IppStatus ippsLogGaussMax_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussMax_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussMax_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);

IppStatus ippsLogGaussMax_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);

IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int
    height, Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
    Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst,
    int height, Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the observation vector [<i>height</i> * <i>step</i>].
<i>mSrc</i>	Pointer to the observation matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in the observation vector <i>pSrc</i> .
<i>pMean</i>	Pointer to the mean vector [<i>width</i>].
<i>pVar</i>	Pointer to the variance vector [<i>width</i>].
<i>width</i>	Length of the mean and variance vectors.
<i>pSrcDst</i>	Pointer to the likelihood vector [<i>height</i>].
<i>height</i>	Length of the vector <i>pSrcDst</i> .

<i>val</i>	Gaussian constant.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussMax` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector `pSrcDst` using the “maximum” operation.

For functions without the `IdVar` suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional `D2` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, \quad$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]), \quad 0 \leq i < height.$$

For functions with the additional `D2L` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, \quad$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]), \quad 0 \leq i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional `D2` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2, \quad$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]), \quad 0 \leq i < height.$$

For functions with the additional `D2L` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2, \quad$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]), \quad 0 \leq i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , <i>pMean</i> , <i>pVar</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

LogGaussMaxMultiMix

Calculate the likelihood probability for multiple Gaussian mixture components, using the maximum operation.

Syntax

```
IppStatus ippLogGaussMaxMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
    Ipp32s* pSrcDst, int height, int scaleFactor);

IppStatus ippLogGaussMaxMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s* pVal, Ipp32s*
    pSrcDst, int height, int scaleFactor);

IppStatus ippLogGaussMaxMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
    Ipp32f* pSrcDst, int height, int scaleFactor);

IppStatus ippLogGaussMaxMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal, Ipp32f*
    pSrcDst, int height, int scaleFactor);
```

```

IppStatus ippsLogGaussMaxMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
    pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f*
    pSrcDst, int height);

IppStatus ippsLogGaussMaxMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
    mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f* pSrcDst,
    int height);

IppStatus ippsLogGaussMaxMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
    pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f*
    pSrcDst, int height);

IppStatus ippsLogGaussMaxMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
    mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f* pSrcDst,
    int height);

IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2Sfs(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
    Ipp32s* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2LSfs(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s* pVal, Ipp32s*
    pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2(const Ipp16s* pMean,
    const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f*
    pVal, Ipp32f* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mMean,
    const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
    Ipp32f* pSrcDst, int height, int scaleFactor);

```

Parameters

<i>pMean</i>	Pointer to the mean vector [<i>height</i> * <i>step</i>].
<i>pVar</i>	Pointer to the variance vector [<i>height</i> * <i>step</i>].
<i>mMean</i>	Pointer to the mean matrix [<i>height</i>][<i>width</i>].
<i>mVar</i>	Pointer to the variance matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in the mean and variance vectors.
<i>pSrc</i>	Pointer to the input vector [<i>width</i>].
<i>width</i>	Number of columns in the mean and variance matrices.
<i>pVal</i>	Pointer to the weight constant vector [<i>height</i>].
<i>pSrcDst</i>	Pointer to the likelihood vector [<i>height</i>].

<i>height</i>	Number of rows in the mean and variance matrices.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussMaxMultiMix` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector `pSrcDst` using the “maximum” operation. The covariance matrix is assumed to be inverse diagonal.

For functions with the `D2` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2, \\ pSrcDst[i] = \max(pSrcDst[i], V[i]), \quad 0 \leq i < height.$$

For functions with the `D2L` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2, \\ pSrcDst[i] = \max(pSrcDst[i], V[i]), \quad 0 \leq i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mMean</i> , <i>mVar</i> , <i>pMean</i> , <i>pVar</i> , <i>pVal</i> , or <i>pSrcDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

LogGaussAdd

Calculates the likelihood probability for multiple observation vectors.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussAdd_Scaled_16s32f_D2(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussAdd_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f
    val, int scaleFactor);

IppStatus ippsLogGaussAdd_32f_D2(const Ipp32f** pSrc, int step, const Ipp32f*
    pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussAdd_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussAdd_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
    pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f
    val);

IppStatus ippsLogGaussAdd_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
    const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f val);

IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
    height, Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val,
    int scaleFactor);

IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val,
    int scaleFactor);

IppStatus ippsLogGaussAdd_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
```

```

IppStatus ippsLogGaussAdd_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f
    val, int scaleFactor);
IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
    scaleFactor);

```

Parameters

<i>pSrc</i>	Pointer to the observation vector [<i>height</i> * <i>step</i>].
<i>mSrc</i>	Pointer to the observation matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in the observation vector <i>pSrc</i> .
<i>pMean</i>	Pointer to the mean vector [<i>width</i>].
<i>pVar</i>	Pointer to the variance vector [<i>width</i>].
<i>width</i>	Length of the mean and variance vectors.
<i>pSrcDst</i>	Pointer to the likelihood vector [<i>height</i>].
<i>height</i>	Number of observation vectors.
<i>val</i>	Weight constant.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussAdd` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector *pSrcDst*.

For functions without the `IdVar` suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional `D2` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, \quad$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}), \quad 0 \leq i < height.$$

For functions with the additional `D2L` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, \quad$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}), \quad 0 \leq i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional `D2` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2, \quad$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}), \quad 0 \leq i < height.$$

For functions with the additional `D2L` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2, \quad$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}), \quad 0 \leq i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , <i>pMean</i> , <i>pVar</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

LogGaussAddMultiMix

*Calculates the likelihood probability
for multiple Gaussian mixture components.*

Syntax

```
IppStatus ippLogGaussAddMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f pVal,
    Ipp32f* pSrcDst, int height, int scaleFactor);

IppStatus ippLogGaussAddMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal, Ipp32f*
    pSrcDst, int height, int scaleFactor);

IppStatus ippLogGaussAddMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
    pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f*
    pSrcDst, int height);

IppStatus ippLogGaussAddMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
    mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f* pSrcDst,
    int height);

IppStatus ippLogGaussAddMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
    pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f*
    pSrcDst, int height);

IppStatus ippLogGaussAddMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
    mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f* pSrcDst,
    int height);

IppStatus ippLogGaussAddMultiMix_LowScaled_16s32f_D2(const Ipp16s* pMean,
    const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f*
    pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussAddMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mMean,
const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
Ipp32f* pSrcDst, int height, int scaleFactor);
```

Parameters

<i>pMean</i>	Pointer to the mean vector [<i>height*step</i>].
<i>pVar</i>	Pointer to the variance vector [<i>height*step</i>].
<i>mMean</i>	Pointer to the mean matrix [<i>height</i>][<i>width</i>].
<i>mVar</i>	Pointer to the variance matrix [<i>height</i>][<i>width</i>].
<i>step</i>	Row step in mean and variance vectors (in <i>pMean</i> elements).
<i>pSrc</i>	Pointer to the input vector [<i>width</i>].
<i>width</i>	Length of the mean and variance matrices.
<i>pVal</i>	Pointer to the weight constant vector [<i>height</i>].
<i>pSrcDst</i>	Pointer to the likelihood vector [<i>height</i>].
<i>height</i>	Number of Gaussian mixture components.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussAddMultiMix` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector *pSrcDst*. The covariance matrix is assumed to be inverse diagonal.

For functions with the `D2` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2, \quad$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}), \quad 0 \leq i < height.$$

For functions with the `D2L` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2, \quad$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{v[i]}) , 0 \leq i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, *pVal*, or *pSrcDst* pointer is NULL.

`ippStsSizeErr` Indicates an error when *width* or *height* is less than or equal to 0.

`ippStsStrideErr` Indicates an error when *step* is less than *width*.

LogGaussMixture

*Calculates the likelihood probability
for the Gaussian mixture.*

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippLogGaussMixture_Scaled_16s32f_D2(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int width, const
    Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);

IppStatus ippLogGaussMixture_Scaled_16s32f_D2L(const Ipp16s* pSrc, const
    Ipp16s** mMean, const Ipp16s** mVar, int height, int width, const Ipp32f*
    pVal, Ipp32f* pResult, int scaleFactor);

IppStatus ippLogGaussMixture_LowScaled_16s32f_D2(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int width, const
    Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
```

```

IppStatus ippsLogGaussMixture_LowScaled_16s32f_D2L(const Ipp16s* pSrc, const
    Ipp16s** mMean, const Ipp16s** mVar, int height, int width, const Ipp32f*
    pVal, Ipp32f* pResult, int scaleFactor);

IppStatus ippsLogGaussMixture_32f_D2(const Ipp32f* pSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int height, int step, int width, const Ipp32f* pVal,
    Ipp32f* pResult);

IppStatus ippsLogGaussMixture_32f_D2L(const Ipp32f* pSrc, const Ipp32f** mMean,
    const Ipp32f** mVar, int height, int width, const Ipp32f* pVal, Ipp32f*
    pResult);

IppStatus ippsLogGaussMixture_64f_D2(const Ipp64f* pSrc, const Ipp64f* pMean,
    const Ipp64f* pVar, int height, int step, int width, const Ipp64f* pVal,
    Ipp64f* pResult);

IppStatus ippsLogGaussMixture_64f_D2L(const Ipp64f* pSrc, const Ipp64f** mMean,
    const Ipp64f** mVar, int height, int width, const Ipp64f* pVal, Ipp64f*
    pResult);

```

Case 2: Operation for the identity covariance matrix

```

IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, const
    Ipp16s* pMean, int height, int step, int width, const Ipp32f* pVal, Ipp32f*
    pResult, int scaleFactor);

IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2L(const Ipp16s* pSrc, const
    Ipp16s** mMean, int height, int width, const Ipp32f* pVal, Ipp32f* pResult,
    int scaleFactor);

IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc,
    const Ipp16s* pMean, int height, int step, int width, const Ipp32f* pVal,
    Ipp32f* pResult, int scaleFactor);

IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2L(const Ipp16s* pSrc,
    const Ipp16s** mMean, int height, int width, const Ipp32f* pVal, Ipp32f*
    pResult, int scaleFactor);

IppStatus ippsLogGaussMixture_IdVar_32f_D2(const Ipp32f* pSrc, const Ipp32f*
    pMean, int height, int step, int width, const Ipp32f* pVal, Ipp32f*
    pResult);

IppStatus ippsLogGaussMixture_IdVar_32f_D2L(const Ipp32f* pSrc, const Ipp32f**
    mMean, int height, int width, const Ipp32f* pVal, Ipp32f* pResult);

IppStatus ippsLogGaussMixture_IdVar_64f_D2(const Ipp64f* pSrc, const Ipp64f*
    pMean, int height, int step, int width, const Ipp64f* pVal, Ipp64f*
    pResult);

IppStatus ippsLogGaussMixture_IdVar_64f_D2L(const Ipp64f* pSrc, const Ipp64f**
    mMean, int height, int width, const Ipp64f* pVal, Ipp64f* pResult);

```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>width</i>].
<i>pMean</i>	Pointer to the mean vector [<i>height</i> * <i>step</i>].
<i>pVar</i>	Pointer to the variance vector [<i>height</i> * <i>step</i>].
<i>mMean</i>	Pointer to the mean matrix [<i>height</i>][<i>width</i>].
<i>mVar</i>	Pointer to the variance matrix [<i>height</i>][<i>width</i>].
<i>height</i>	Number of Gaussian mixture components.
<i>step</i>	Row step in mean and variance vectors (in <i>pMean</i> elements).
<i>width</i>	Length of the mean and variance matrix rows; also length of the <i>pSrc</i> vector.
<i>pVal</i>	Pointer to the vector of weight constants [<i>height</i>].
<i>pResult</i>	Pointer to the output mixture value.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussMixture` is declared in the `ippsr.h` file. This function calculates observation probability for the Gaussian mixture and returns it in *pResult*.

For functions that do not have the `IdVar` suffix in their names, the covariance matrix is assumed to be inverse diagonal.

For these functions that also have the additional `D2` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2,$$

for $i = 0, \dots, height - 1$, and

$$pResult[0] = \ln \sum_{i=0}^{height-1} e^{V[i]} .$$

For these functions with the `D2L` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 ,$$

$$i = 0, \dots, height - 1 .$$

For functions that have the `IdVar` suffix in their names, the covariance matrix is assumed to be identity.

For these functions that have the additional `D2` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[j] - pMean[i \cdot step + j])^2 ,$$

for $i = 0, \dots, height - 1$, and

$$pResult[0] = \ln \sum_{i=0}^{height-1} e^{V[i]}$$

For these functions with the additional `D2L` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[j] - mMean[i][j])^2 , \quad i = 0, \dots, height - 1 .$$

Function flavors that perform integer scaling (distinguished by the presence of `scaleFactor` argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , <i>pMean</i> , <i>mMean</i> , <i>pVar</i> , <i>mVar</i> , <i>pVal</i> , or <i>pResult</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

LogGaussMixtureSelect

Calculates the likelihood probability for the Gaussian mixture using Gaussian selection.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int step, int width, const Ipp32s* pVal,
    const Ipp8u* pSign, int height, Ipp32s* pResult, int frames, Ipp32f none,
    int scaleFactor);

IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s** mMean, const Ipp16s** mVar, int width, const Ipp32s* pVal,
    const Ipp8u* pSign, int height, Ipp32s* pResult, int frames, Ipp32f none,
    int scaleFactor);

IppStatus ippsLogGaussMixture_Select_32f_D2(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int step, int width, const Ipp32f* pVal, const
    Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f none);

IppStatus ippsLogGaussMixture_Select_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f** mMean, const Ipp32f** mVar, int width, const Ipp32f* pVal, const
    Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f none);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2(const Ipp16s* pSrc,
    const Ipp16s* pMean, int step, int width, const Ipp32s* pVal, const Ipp8u*
    pSign, int height, Ipp32s* pResult, int frames, Ipp32f none, int
    scaleFactor);

IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s** mMean, int width, const Ipp32s* pVal, const Ipp8u* pSign, int
    height, Ipp32s* pResult, int frames, Ipp32f none, int scaleFactor);
```



```
IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2(const Ipp32f* pSrc, const
    Ipp32f* pMean, int step, int width, const Ipp32f* pVal, const Ipp8u* pSign,
    int height, Ipp32f* pResult, int frames, Ipp32f none);

IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f** mMean, int width, const Ipp32f* pVal, const Ipp8u* pSign, int
    height, Ipp32f* pResult, int frames, Ipp32f none);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>frames*step</i>].
<i>pMean</i>	Pointer to the mean vector [<i>height*step</i>].
<i>pVar</i>	Pointer to the variance vector [<i>height*step</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>frames</i>][<i>width</i>].
<i>mMean</i>	Pointer to the mean matrix [<i>height</i>][<i>width</i>].
<i>mVar</i>	Pointer to the variance matrix [<i>height</i>][<i>width</i>].
<i>pVal</i>	Pointer to the vector of weight constants [<i>height</i>].
<i>pSign</i>	Pointer to the vector of Gaussian calculation signs [<i>frames*(height+7)/8</i>] (in bytes).
<i>step</i>	Row step in mean, variance and input vectors (in <i>pMean</i> elements).
<i>width</i>	Length of the mean, variance and input matrix rows.
<i>height</i>	Number of Gaussian mixture components.
<i>pResult</i>	Pointer to the vector of output mixture values [<i>frames</i>].
<i>frames</i>	Number of input vectors; also the length of <i>pSign</i> row.
<i>none</i>	Result value if no Gaussian is calculated.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

Discussion

The function `ippsLogGaussMixture_Select` is declared in the `ippsr.h` file. This function calculates observation probabilities for the Gaussian mixture and *frames* input vectors and returns them in *pResult* vector. Gaussians with corresponding signs equal to zero are not calculated. If no Gaussian is calculated for the input vector, *none* value is returned for this vector.

Let $s(i,k)$ be the k -th bit of the i -th row of *pSign*.

For functions that do not have the `IdVar` suffix in their names, the covariance matrix is assumed to be inverse diagonal.

For these functions that also have the additional `D2` suffix,

$$V[i, k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[k \cdot step + j] - pMean[i \cdot step + j])^2$$

for $k = 0, \dots, frames - 1$, $i = 0, \dots, height - 1$ and $s(i, k) \neq 0$,

and

$$pResult[k] = \begin{cases} none, & \text{if } s(i, k) = 0 \text{ for } i = 0, \dots, height - 1 \\ \ln \sum_{\substack{i=0 \\ s(i, k) \neq 0}}^{height-1} e^{V[i, k]}, & \text{otherwise} \end{cases}$$

For these functions with the `D2L` suffix,

$$V[i, k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (mSrc[k][j] - mMean[i][j])^2$$

for $k = 0, \dots, frames - 1$, $i = 0, \dots, height - 1$ and $s(i, k) \neq 0$,

and the `pResult` vector is computed by the same formula as given above.

For functions that have the `IdVar` suffix in their names, the covariance matrix is assumed to be identity.

For these functions that also have the additional `D2` suffix,

$$V[i, k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[k \cdot step + j] - pMean[i \cdot step + j])^2$$

for $k = 0, \dots, frames - 1$, $i = 0, \dots, height - 1$ and $s(i, k) \neq 0$,

and the `pResult` vector is computed by the same formula as given above.

For these functions with the `D2L` suffix,

$$V[i, k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[k][j] - mMean[i][j])^2$$

for $k = 0, \dots, \text{frames} - 1$, $i = 0, \dots, \text{height} - 1$ and $s(i, k) \neq 0$,
and the *pResult* vector is computed by the same formula as given above.

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-\text{scaleFactor}}$ to intermediate sums in the above formulae, rather than to output results.

Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when <i>pSrc</i> , <i>pMean</i> , <i>pVar</i> , <i>mSrc</i> , <i>mMean</i> , <i>mVar</i> , <i>pSign</i> , <i>pVal</i> or <i>pResult</i> pointer is NULL.
<i>ippStsSizeErr</i>	Indicates an error when <i>width</i> , <i>height</i> , or <i>frames</i> is less than or equal to 0.
<i>ippStsStrideErr</i>	Indicates an error when <i>step</i> is less than <i>width</i> .
<i>ippStsNoGaussian</i>	Indicates a warning when no Gaussian is calculated for one of the input vectors.

BuildSignTable

Fills sign table for Gaussian mixture calculation.

Syntax

```
IppStatus ippBuildSignTable_8u16u(const Ipp32s* pIndx, int num, const Ipp8u**
    mShortlist, int clust, int width, int shift, Ipp8u* pSign, int frames, int
    comps);
```

```
IppStatus ippBuildSignTable_Var_8u16u(const Ipp32s* pIndx, const int* pNum,
    const Ipp8u** mShortlist, int clust, int width, int shift, Ipp8u* pSign, int
    frames, int comps);
```

Parameters

<i>pIndx</i>	Pointer to the cluster indexes vector ($[\text{frames} * \text{num}]$ or sum of <i>pNum</i> elements).
<i>num</i>	Number of clusters for each input vector.
<i>pNum</i>	Number of clusters vector $[\text{frames}]$.

<i>mShortlist</i>	Pointer to the shortlist matrix [<i>clust</i> * <i>width</i>] (in bytes).
<i>clust</i>	Number of rows in shortlist vector (equal to the codebook size).
<i>width</i>	Row length in shortlist matrix in bytes.
<i>shift</i>	First element displacement in shortlist vector rows (in bits).
<i>pSign</i>	Pointer to the output vector of Gaussian calculation signs [<i>frames</i> *(<i>comps</i> +7)/8] (in bytes).
<i>frames</i>	Number of input vectors (rows in <i>pSign</i> vector).
<i>comps</i>	Number of Gaussian mixture components (bit columns in <i>pSign</i> vector).

Discussion

The functions `ippsBuildSignTable` are declared in the `ippsr.h` file. These functions build sign table for Gaussian mixture calculation using Gaussian selection technique. The sign table is used as the input argument for `ippsLogGaussMixture_Select` functions. The input vectors could activate one or several codebook clusters. Shortlist table rows correspond to clusters, and its columns – to Gaussians. Row substring of length *clust* indicates if the cluster is active for a mixture component.

Let $l(i) = i \cdot \text{num}$ for `ippsBuildSignTable` function, and

$$l(i) = \sum_{j=0}^{i-1} pNum[j] \text{ for } ippsBuildSignTable_Var \text{ function,}$$

for $i = 0, \dots, frames - 1$.

Let also c_k be the bit substring of the k -th row of *mShortlist*, consisting of bits *shift*,...*shift*+*comps*-1, for $k = 0, \dots, clust - 1$.

Then elements $l(i), \dots, l(i+1)-1$ of the *pIndx* vector are active cluster numbers for i -th input vector. The i -th row of *pSign* is set to the logical sum of substrings

$$c_{l(i)}, \dots, c_{l(i+1)-1}, \quad i = 0, \dots, frames - 1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pIndx</i> , <i>pSign</i> , <i>mShortlist</i> or <i>pNum</i> pointer is NULL.

<code>ippStsSizeErr</code>	Indicates an error when <i>clust</i> , <i>width</i> , <i>frames</i> , <i>comps</i> , <i>num</i> , or one of <i>pNum</i> vector elements is less than or equal to 0, or when <i>shift</i> is less than 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>width</i> is less than $(shift+comps+7)/8$.
<code>ippStsBadArgErr</code>	Indicates an error when one of <i>pIndx</i> vector elements is less than 0, or greater than or equal to <i>clust</i> .

FillShortlist_Row

Fills row-wise shortlist table for Gaussian selection.

Syntax

```
IppStatus ippsFillShortlist_Row_1u(const Ipp32s* pIndx, int height, int num,
    Ipp8u** mShortlist, int clust, int width, int shift);
IppStatus ippsFillShortlist_RowVar_1u(const Ipp32s* pIndx, const int* pNum, int
    height, Ipp8u** mShortlist, int clust, int width, int shift);
```

Parameters

<i>pIndx</i>	Pointer to the cluster indexes vector ($[num*clust]$ or sum of <i>pNum</i> elements).
<i>height</i>	Number of Gaussian mixture components.
<i>num</i>	Number of Gaussian means for each cluster.
<i>pNum</i>	Number of Gaussians vector $[clust]$.
<i>mShortlist</i>	Pointer to the shortlist matrix $[clust*width]$ (in bytes).
<i>clust</i>	Number of rows in shortlist matrix (equal to the codebook size).
<i>width</i>	Row length in shortlist matrix (in bytes).
<i>shift</i>	First element displacement in shortlist matrix rows (in bits).

Discussion

The function `ippsFillShortlist_Row` is declared in the `ippsr.h` file. The function fills the shortlist table and provides that this table contains at least one Gaussian of the mixture for each cluster. This is done by quantizing of cluster centroids on codebook made of Gaussian mixture means. Quantization results are placed to `pIndx` vector.

Bits in the shortlist table `mShortlist` that correspond to one or more cluster centroids nearest to Gaussian mixture means are set to 1. The shortlist is the bit table with elements indicating whether the i -th Gaussian should be calculated if the input vector activates the k -th cluster of the codebook. The number of rows is equal to the codebook size, the number of columns is equal to the overall number of Gaussians in the model. Row step is chosen so as to start each row from the byte boundary.

The shortlist table should be zeroed after definition. Components of mixtures have sequential indexes. The `shift` argument is the index of the first mixture component in the model, and the `height` argument is the number of the components in the mixture.

Let $l(k) = k \cdot \text{num}$ for `ippsFillShortlist_Row` function,

$$l(k) = \sum_{j=0}^{k-1} pNum[j] \text{ for } ippsFillShortlist_RowVar \text{ function,}$$

and $c(k,i)$ be equal to $(shift+i)$ -th bit in the k -th row of `mShortlist`, for $i = 0, \dots, height - 1$, $k = 0, \dots, clust - 1$.

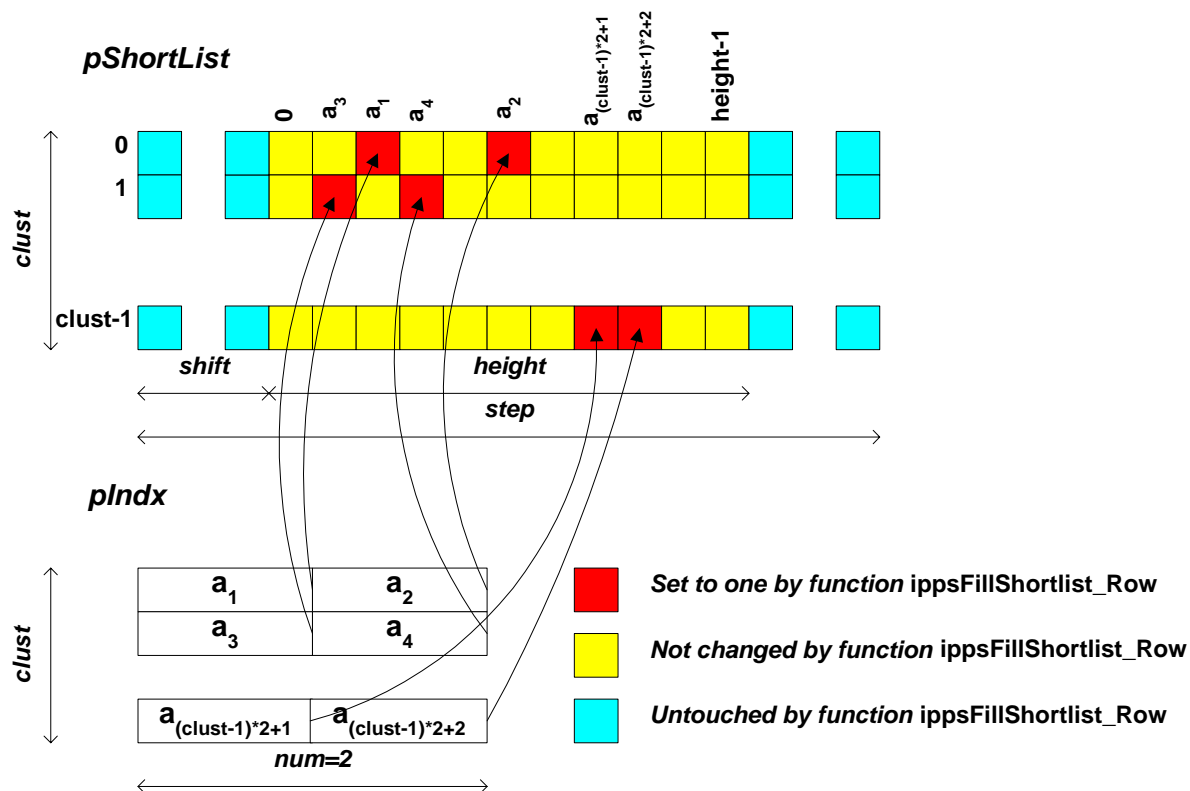
Then `height` columns of the shortlist table are filled as:

$$c(k, pIndx[m]) = 1, \quad m = l(k), \dots, l(k+1)-1, \quad k = 0, \dots, clust - 1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pIndx</code> or <code>pShortlist</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>clust</code> , <code>height</code> , <code>width</code> , <code>num</code> or one of <code>pNum</code> vector elements is less than or equal to 0, or when <code>shift</code> is less than 0.
<code>ippStsStrideErr</code>	Indicates an error when <code>width</code> is less than $(shift+height+7)/8$.
<code>ippStsBadArgErr</code>	Indicates an error when one of <code>pIndx</code> vector elements is less than 0, or greater than or equal to <code>height</code> .

Figure 8-3 Execution of `ippsFillShortlist_Row` function for `num=2`



FillShortlist_Column

Fills column-wise shortlist table for Gaussian selection.

Syntax

```
IppStatus ippsFillShortlist_Column_lu(const Ipp32s* pIndx, int num, Ipp8u**
    mShortlist, int clust, int width, int shift, int height);

IppStatus ippsFillShortlist_ColumnVar_lu(const Ipp32s* pIndx, const int* pNum,
    Ipp8u** mShortlist, int clust, int width, int shift, int height);
```

Parameters

<i>pIdx</i>	Pointer to the cluster indexes vector ($[num * height]$ or sum of <i>pNum</i> elements).
<i>height</i>	Number of Gaussian mixture components.
<i>num</i>	Number of clusters for each Gaussian mean.
<i>pNum</i>	Number of clusters vector $[height]$.
<i>mShortlist</i>	Pointer to the shortlist matrix $[clust * width]$ (in bytes).
<i>clust</i>	Number of rows in shortlist matrix (equal to the codebook size).
<i>width</i>	Row length in shortlist matrix (in bytes).
<i>shift</i>	First element displacement in shortlist matrix rows (in bits).

Discussion

The function `ippsFillShortlist_Column` is declared in the `ippsr.h` file. This function fills the part of the shortlist table corresponding to the Gaussian mixture.

The shortlist is the bit table with elements indicating whether the i -th Gaussian should be calculated if the input vector activates the k -th cluster of the codebook. The number of rows is equal to the codebook size, the number of columns is equal to the overall number of Gaussians in the model. Row step is chosen so as to start each row from the byte boundary.

The shortlist table should be zeroed after definition. Components of mixtures have sequential indexes. The *shift* argument is the index of the first mixture component in the model, and the *height* argument is the number of the components in the mixture.

After means of Gaussian mixture are quantized by one of [ippsVQSingle_Sort](#), [VQSingle_Thresh](#) functions, quantization results can be used to fill the shortlist table. Bits corresponding to active clusters for mixture components are set to 1.

Let $l(i) = i \cdot num$ for `ippsFillShortlist_Column` function,

$$l(i) = \sum_{j=0}^{i-1} pNum[j] \text{ for } ippsFillShortlist_ColumnVar \text{ function,}$$

and $c(k, i)$ be equal to $(shift + i)$ -th bit in the k -th row of *mShortlist*, for $i = 0, \dots, height - 1$, $k = 0, \dots, clust - 1$.

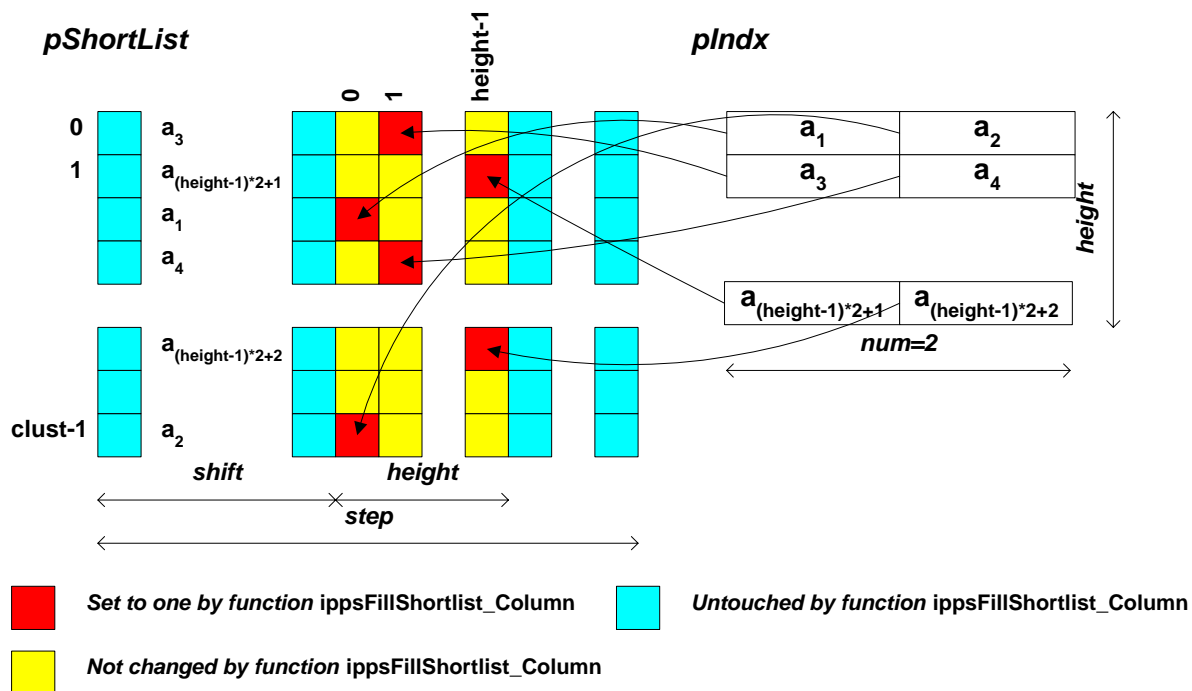
Then *height* columns of the shortlist table are filled as:

$c(pIndx[m], i) = 1$, $m = l(i), \dots l(i+1)-1$, $i = 0, \dots height - 1$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pIndx</code> or <code>pShortlist</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>clust</code> , <code>height</code> , <code>width</code> , <code>num</code> or one of <code>pNum</code> vector elements is less than or equal to 0, or when <code>shift</code> is less than 0.
<code>ippStsStrideErr</code>	Indicates an error when <code>width</code> is less than $(shift+height+7)/8$.
<code>ippStsBadArgErr</code>	Indicates an error when one of <code>pIndx</code> vector elements is less than 0, or greater than or equal to <code>clust</code> .

Figure 8-4 Execution of `ippsFillShortlist_Column` function for `num=2`



DTW

Computes the distance between observation and reference vector sequences using Dynamic Time Warping algorithm.

Syntax

```
IppStatus ippsDTW_L2_8u32s_D2Sfs(const Ipp8u* pSrc1, int height1, const Ipp8u*
    pSrc2, int height2, int width, int step, Ipp32s* pDist, int delta, Ipp32s
    beam, int scaleFactor);

IppStatus ippsDTW_L2_8u32s_D2LSfs(const Ipp8u** mSrc1, int height1, const
    Ipp8u** mSrc2, int height2, int width, Ipp32s* pDist, int delta, Ipp32s
    beam, int scaleFactor);

IppStatus ippsDTW_L2Low_16s32s_D2Sfs(const Ipp16s* pSrc1, int height1, const
    Ipp16s* pSrc2, int height2, int width, int step, Ipp32s* pDist, int delta,
    Ipp32s beam, int scaleFactor);

IppStatus ippsDTW_L2Low_16s32s_D2LSfs(const Ipp16s** mSrc1, int height1, const
    Ipp16s** mSrc2, int height2, int width, Ipp32s* pDist, int delta, Ipp32s
    beam, int scaleFactor);

IppStatus ippsDTW_L2_32f_D2(const Ipp32f* pSrc1, int height1, const Ipp32f*
    pSrc2, int height2, int width, int step, Ipp32f* pDist, int delta, Ipp32f
    beam);

IppStatus ippsDTW_L2_32f_D2L(const Ipp32f** mSrc1, int height1, const Ipp32f**
    mSrc2, int height2, int width, Ipp32f* pDist, int delta, Ipp32f beam);
```

Parameters

<i>pSrc1</i>	Pointer to the first input (observation) vector \mathbf{x} [<i>height1</i> * <i>step</i>].
<i>pSrc2</i>	Pointer to the second input (reference) vector \mathbf{y} [<i>height2</i> * <i>step</i>].
<i>mSrc1</i>	Pointer to the first input (observation) matrix \mathbf{x} [<i>height1</i>][<i>width</i>].
<i>mSrc2</i>	Pointer to the second input (reference) matrix \mathbf{y} [<i>height2</i>][<i>width</i>].
<i>height1</i>	Number of rows in the first input matrix (<i>N1</i>).
<i>height2</i>	Number of rows in the second input matrix (<i>N2</i>).
<i>width</i>	Vector dimension. Equal to the length of the input matrices row (<i>M</i>).
<i>step</i>	Row step in <i>pSrc1</i> and <i>pSrc2</i> .

<i>pDist</i>	Pointer to the distance value.
<i>beam</i>	Beam value, used if positive.
<i>delta</i>	Endpoint constraint relaxation value.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

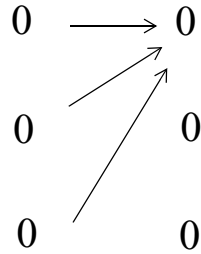
Description

The function `ippsDTW` is declared in the `ippsr.h` file. This function calculates the distance between two vector sequences using the Dynamic Time Warping (DTW) principle. Computations are performed as follows:

$$pDist[0] = \min_W \sum_{i=0}^{N1-1} |\mathbf{x}_i - \mathbf{y}_{W(i)}| ,$$

when $0 \leq W(0) \leq \delta$, $0 \leq W(i) - W(i-1) \leq 2$, $i = 1, \dots, N1-1$, $N2-1-\delta \leq W(N1-1) \leq N2-1$.

Local constraint on DTW path search looks as follows:



To compute distance between vectors *a* and *b*, the function uses L2-norm as given by:

$$|a - b| = \sqrt{\sum_{j=0}^{M-1} (a(j) - b(j))^2}$$

If the *beam* value is positive, paths *v* that are outside the beam, i.e., satisfy the condition

$$\sum_{i=0}^k |\mathbf{x}_i - \mathbf{y}_{V(i)}| > \min_W \sum_{i=0}^k |\mathbf{x}_i - \mathbf{y}_{W(i)}| - beam , \quad k = 0, \dots, N1-1$$

are pruned away.

The global constraints on path search are described by the following formulas.

The upper boundaries are:

$$J=2i-1+\delta; \quad J=i/2-(N1/2-N2)+\delta;$$

The lower boundaries are:

$$J=i/2+1/2-\delta; \quad J=2i-(2*N1-N2)-\delta.$$

Note that if $height1 < height2$, then $pSrc1$ interchanges with $pSrc2$, that is, $pSrc1$ becomes the reference vector and $pSrc2$ is assumed to be the test vector.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc1</code> , <code>pSrc2</code> , <code>mSrc1</code> , <code>mSrc2</code> , or <code>pDist</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>height1</code> , <code>height2</code> , or <code>width</code> is less than or equal to 0, or <code>delta</code> is less than 0 or greater than <code>height2</code> .
<code>ippStsStrideErr</code>	Indicates an error when <code>step</code> is less than <code>width</code> .
<code>ippStsNoOperation</code>	Indicates that there are now admissible paths for <code>height1</code> , <code>height2</code> and <code>delta</code> values.

Model Estimation

This section describes functions that are needed to estimate the parameters of the acoustic and language models.

Some of these functions are used in the Intel IPP Speech recognition Sample. See *Gaussian Calculations* sample downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

MeanColumn

Computes the mean values for the column elements.

Syntax

```
IppStatus ippsMeanColumn_16s_D2(const Ipp16s* pSrc, int height, int step,
    Ipp16s* pDstMean, int width);
IppStatus ippsMeanColumn_16s_D2L(const Ipp16s** mSrc, int height, Ipp16s*
    pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
    Ipp32f* pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
    pDstMean, int width);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height</i> * <i>step</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>step</i>	Row step in the input vector (measured in <i>pSrc</i> elements).
<i>pDstMean</i>	Pointer to the output mean vector [<i>width</i>].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> , and also the length of the output mean vector <i>pDstMean</i> .

Description

The function `ippsMeanColumn` is declared in the `ippsr.h` file. This function calculates the mean values for the column elements of the input matrix as follows:

For functions with the D2 suffix,

$$pDstMean[j] = \frac{1}{height} \sum_{i=1}^{height-1} pSrc[i \cdot step + j], \quad 0 \leq j < width$$

For functions with the D2L suffix,

$$pDstMean[j] = \frac{1}{height} \sum_{i=1}^{height-1} mSrc[i][j], \quad 0 \leq j < width$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> , <code>mSrc</code> , or <code>pDstMean</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>height</code> or <code>width</code> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <code>step</code> is less than <code>width</code> .

VarColumn

Calculates the variances for the column elements.

Syntax

```
IppStatus ippVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int step,
    Ipp16s* pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height, Ipp16s*
    pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
    Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);
IppStatus ippVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
    pSrcMean, Ipp32f* pDstVar, int width);
```

Parameters

<code>pSrc</code>	Pointer to the input vector [<code>height*step</code>].
<code>mSrc</code>	Pointer to the input matrix [<code>height</code>][<code>width</code>].
<code>height</code>	Number of rows in the input matrix <code>mSrc</code> .
<code>step</code>	Row step in the input vector <code>pSrc</code> .
<code>pSrcMean</code>	Pointer to the input mean vector [<code>width</code>].
<code>pDstVar</code>	Pointer to the output variance vector [<code>width</code>].
<code>width</code>	Number of columns in the input matrix <code>mSrc</code> , and also the length of the input mean vector <code>pSrcMean</code> and the output variance vector <code>pDstVar</code> .

scaleFactor Scale factor, refer to [“Integer Scaling”](#) in Chapter 2.

Description

The function `ippsVarColumn` is declared in the `ippsr.h` file. This function calculates the variances for the column elements of the matrix as follows:

For functions with the D2 suffix,

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} pSrc[i \cdot step + j]^2}{height - 1},$$

$0 \leq j < width$.

For functions with the D2L suffix

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} mSrc[i][j]^2}{height - 1},$$

$0 \leq j < width$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , <i>pSrcMean</i> , or <i>pDstVar</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> is less than or equal to 0 or <i>height</i> is less than or equal to 1.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

MeanVarColumn

Calculates the means and variances for the column elements of a matrix.

Syntax

```
IppStatus ippsMeanVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int step,
    Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height, Ipp16s*
    pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int height,
    int step, Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height,
    Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
    Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsMeanVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
    pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsMeanVarColumn_16s16s32s_D2(const Ipp16s* pSrc, int height, int
    step, Ipp16s* pDstMean, Ipp32s* pDstVar, int width);
IppStatus ippsMeanVarColumn_16s16s32s_D2L(const Ipp16s** mSrc, int height,
    Ipp16s* pDstMean, Ipp32s* pDstVar, int width);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height</i> * <i>step</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pDstMean</i>	Pointer to the output mean vector [<i>width</i>].
<i>pDstVar</i>	Pointer to the output variance vector [<i>width</i>].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> , and also the length of the output mean vector <i>pDstMean</i> and variance vector <i>pDstVar</i> .

scaleFactor Scale factor, refer to “[Integer Scaling](#)” in Chapter 2. *scaleFactor* argument (and integer scaling) is used for *pDstVar* only, while *pDstMean* elements are not scaled.

Discussion

The function `ippsMeanVarColumn` is declared in the `ippsr.h` file. This function calculates both the means and the variances for the column elements of the input matrix. See functions [ippsMeanColumn](#) and [ippsVarColumn](#) for calculation details.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>mSrc</i> , <i>pDstMean</i> , or <i>pDstVar</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> is less than or equal to 0 or <i>height</i> is less than or equal to 1.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

WeightedMeanColumn

Computes the weighted mean values for the column elements.

Syntax

```
IppStatus ippsWeightedMeanColumn_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pWgt, int height, Ipp32f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pWgt, int height, Ipp32f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pWgt, int height, Ipp64f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pWgt, int height, Ipp64f* pDstMean, int width);
```

Parameters

pSrc Pointer to the input vector [*height***step*].

<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].
<i>pWgt</i>	Pointer to the weight vector [<i>height</i>].
<i>height</i>	Number of rows in the input matrix.
<i>step</i>	Row step in the input vector (measured in <i>pSrc</i> elements).
<i>pDstMean</i>	Pointer to the output mean vector [<i>width</i>].
<i>width</i>	Number of columns in the input matrix, and also the length of the output mean vector <i>pDstMean</i> .

Description

The function `ippsWeightedMeanColumn` is declared in the `ippsr.h` file. This function calculates the weighted mean values for the column elements of the input matrix as follows:
For functions with the D2 suffix,

$$pDstMean[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot pSrc[i \cdot step + j], 0 \leq j < width.$$

For functions with the D2L suffix,

$$pDstMean[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot mSrc[i][j], 0 \leq j < width.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , <i>pWgt</i> , or <i>pDstMean</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

WeightedVarColumn

*Computes the weighted variance values
for the column elements.*

Syntax

```
IppStatus ippsWeightedVarColumn_32f_D2(const Ipp32f* pSrc, int step, const  
    Ipp32f* pWgt, int height, const Ipp32f* pSrcMean, Ipp32f* pDstVar, int  
    width);  
  
IppStatus ippsWeightedVarColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*  
    pWgt, int height, const Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);  
  
IppStatus ippsWeightedVarColumn_64f_D2(const Ipp64f* pSrc, int step, const  
    Ipp64f* pWgt, int height, const Ipp64f* pSrcMean, Ipp64f* pDstVar, int  
    width);  
  
IppStatus ippsWeightedVarColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*  
    pWgt, int height, const Ipp64f* pSrcMean, Ipp64f* pDstVar, int width);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height</i> * <i>step</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].
<i>pWgt</i>	Pointer to the weight vector [<i>height</i>].
<i>height</i>	Number of rows in the input matrix.
<i>step</i>	Row step in the input vector (measured in <i>pSrc</i> elements).
<i>pSrcMean</i>	Pointer to the input mean vector [<i>width</i>].
<i>pDstVar</i>	Pointer to the output variance vector [<i>width</i>].
<i>width</i>	Number of columns in the input matrix, and also the length of the input mean vector <i>pSrcMean</i> and the output variance vector <i>pDstVar</i> .

Description

The function `ippsWeightedVarColumn` is declared in the `ippsr.h` file. This function calculates the weighted variance values for the column elements of the input matrix as follows:
For functions with the D2 suffix,

$$pDstVar[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot pSrc[i \cdot step + j]^2 - pSrcMean[j]^2, 0 \leq j < width.$$

For functions with the D2L suffix,

$$pDstVar[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot mSrc[i][j]^2 - pSrcMean[j]^2, 0 \leq j < width.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , <i>pWgt</i> , <i>pSrcMean</i> , or <i>pDstVar</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

WeightedMeanVarColumn

Computes weighted mean and variance values for the column elements.

Syntax

```
IppStatus ippsWeightedMeanVarColumn_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pWgt, int height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pWgt, int height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pWgt, int height, Ipp64f* pDstMean, Ipp64f* pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pWgt, int height, Ipp64f* pDstMean, Ipp64f* pDstVar, int width);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>height</i> * <i>step</i>].
<i>mSrc</i>	Pointer to the input matrix [<i>height</i>][<i>width</i>].

<i>pWgt</i>	Pointer to the weight vector [<i>height</i>].
<i>height</i>	Number of rows in the input matrix.
<i>step</i>	Row step in the input vector (measured in <i>pSrc</i> elements).
<i>pDstMean</i>	Pointer to the output mean vector [<i>width</i>].
<i>pDstVar</i>	Pointer to the output variance vector [<i>width</i>].
<i>width</i>	Number of columns in the input matrix, and also the length of the output mean vector <i>pSrcMean</i> and the output variance vector <i>pDstVar</i> .

Description

The function `ippsWeightedMeanVarColumn` is declared in the `ippsr.h` file. This function calculates both weighted means and weighted variances for the column elements of the input matrix. See functions [ippsWeightedMeanColumn](#) and [ippsWeightedVarColumn](#) for calculation details.

The code example below shows how you can use the function `ippsWeightedMeanVarColumn`.

Example 8-2 Weights, Means and Variances EM Re-Estimation

```

/* Input:  int height;           // mixture components number
           int width;           // observation space dimension
           int step;            // row step for mean, var and obs (step>=width)
           int num;            // observations number
           int step1;          // row step for gamma (step1>=num)
           float obs [num*step] // observation vectors
Update:   float weight[height]  // Gaussian weights
           float mean[height*step] // Gaussian mean vectors
           float var [height*step] // Gaussian variance vectors
Output:   float result;         // probability logarithms sum */
{
    float gamma [height*step1] // gamma matrix
    float gammaT [height]      // gamma sums vector
    int k; float sum, sumGamma;
    /* adjust determinants for probability calculation */
    ippsLn_32f_I(weight,height);
    for (k=0; k<height; k++) {
        ippsSumLn_32f(var+k*step,width,&sum);
        weight[k]+=0.5f*(sum-width*log(2.0*3.1415926));
    }

    /* invert variances for probability calculation */
    ippsDivCRev_32f_I(var,height*step);
    /* logarithm of weighted Gaussian probabilities */
    ippsLogGauss_32f_D2(obs,step,mean,var,width,gamma,num,weight[0]);
    ippsCopy_32f(gamma,probs,num);
    for (k=1; k<height; k++) {
        ippsLogGauss_32f_D2(obs,step,mean+k*step,var+k*step,width,
                           gamma+k*step1,num,weight[k]);
        ippsLogAdd_32f(gamma+k*step1,probs,num,ippAlgHintNone);
    }
    ippsSum_32f(probs,num,&result,ippAlgHintNone);
    /* gamma matrix and sum calculation */
    ippsExp_32f_I(gamma,height*step1);

```

Example 8-2 Weights, Means and Variances EM Re-Estimation

```

        ippSumRow_32f_D2(gamma,height,step1,gammaT,num);
        ippSum_32f(gammaT,height,&sumGamma,ippAlgHintNone);
        /* weights update */
        ippDivC_32f(gammaT,sumGamma,weight,height);
        /* means and variances update */
        for (k=0; k<height; k++) {
            ippDivC_32f_I(gammaT[k],gamma+k*step1,num);
            ippWeightedMeanVarColumn_32f_D2(obs,step,gamma+k*step1,num,
                                           mean+k*step,var+k*step,width);
        }
    }
}

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>mSrc</i> , <i>pWgt</i> , <i>pDstMean</i> , or <i>pDstVar</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

NormalizeColumn

*Normalizes the matrix columns
given the column means and variances.*

Syntax

```

IppStatus ippNormalizeColumn_16s_D2Sfs(Ipp16s* pSrcDst, int step, int height,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);
IppStatus ippNormalizeColumn_16s_D2LSfs(Ipp16s** mSrcDst, int height, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);
IppStatus ippNormalizeColumn_32f_D2(Ipp32f* pSrcDst, int step, int height,
    const Ipp32f* pMean, const Ipp32f* pVar, int width);
IppStatus ippNormalizeColumn_32f_D2L(Ipp32f** mSrcDst, int height, const
    Ipp32f* pMean, const Ipp32f* pVar, int width);

```

Parameters

<i>pSrcDst</i>	Pointer to the input and output vector [<i>height</i> * <i>step</i>].
<i>mSrcDst</i>	Pointer to the input and output matrix [<i>height</i>][<i>width</i>].
<i>pMean</i>	Pointer to the column mean vector [<i>width</i>].
<i>pVar</i>	Pointer to the column variance vector [<i>width</i>].
<i>width</i>	Length of the mean and variance vectors.
<i>step</i>	Row step in the input and output vector <i>pSrcDst</i> .
<i>height</i>	Number of rows in the input and output matrix <i>mSrcDst</i> .

Description

The function `ippsNormalizeColumn` is declared in the `ippsr.h` file. This function normalizes the columns of the matrix *mSrcDst* as follows:

For functions with the D2 suffix,

$$pSrcDst[i \cdot step + j] = (pSrcDst[i \cdot step + j] - pMean[j]) \cdot pVar[j] ,$$

$$0 \leq i < height, 0 \leq j < width.$$

For functions with the D2L suffix,

$$mSrcDst[i][j] = (mSrcDst[i][j] - pMean[j]) \cdot pVar[j] ,$$

$$0 \leq i < height, 0 \leq j < width.$$

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <i>pSrcDst</i> , <i>mSrcDst</i> , <i>pMean</i> , or <i>pVar</i> pointer is NULL.
<code>ippsStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippsStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

NormalizeInRange

Normalizes and scales input vector elements.

Syntax

```
IppStatus ippsNormalizeInRange_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len,
    Ipp32f lowCut, Ipp32f highCut, Ipp8u range);

IppStatus ippsNormalizeInRange_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp32f lowCut, Ipp32f highCut, Ipp16s range);

IppStatus ippsNormalizeInRange_16s_I(Ipp16s* pSrcDst, int len, Ipp32f lowCut,
    Ipp32f highCut, Ipp16s range);

IppStatus ippsNormalizeInRange_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len,
    Ipp32f lowCut, Ipp32f highCut, Ipp8u range);

IppStatus ippsNormalizeInRange_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int
    len, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);

IppStatus ippsNormalizeInRange_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f lowCut, Ipp32f highCut, Ipp32f range);

IppStatus ippsNormalizeInRange_32f_I(Ipp32f* pSrcDst, int len, Ipp32f lowCut,
    Ipp32f highCut, Ipp32f range);

IppStatus ippsNormalizeInRangeMinMax_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int
    len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp8u
    range);

IppStatus ippsNormalizeInRangeMinMax_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s
    range);

IppStatus ippsNormalizeInRangeMinMax_16s_I(Ipp16s* pSrcDst, int len, Ipp16s
    valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);

IppStatus ippsNormalizeInRangeMinMax_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int
    len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp8u
    range);

IppStatus ippsNormalizeInRangeMinMax_32f16s(const Ipp32f* pSrc, Ipp16s* pDst,
    int len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s
    range);

IppStatus ippsNormalizeInRangeMinMax_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp32f
    range);
```

```
IppStatus ippsNormalizeInRangeMinMax_32f_I(Ipp32f* pSrcDst, int len, Ipp32f
    valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp32f range);
```

Parameters

<i>pSrc</i>	Pointer to the input array [<i>len</i>].
<i>pSrcDst</i>	Pointer to the input and output array (for the in-place operation) [<i>len</i>].
<i>pDst</i>	Pointer to the output array [<i>len</i>].
<i>len</i>	Number of elements in the input and output array.
<i>lowCut</i>	Lower cutoff value.
<i>highCut</i>	Higher cutoff value.
<i>range</i>	Upper bound of output data values (lower bound is 0).
<i>valMin</i>	Minimum value of input data.
<i>valMax</i>	Maximum value of input data.

Description

The function `ippsNormalizeInRange` is declared in the `ippsr.h` file. This function first normalizes the input vector elements to the range from 0 to 1 imposing lower and higher cutoffs ($0 \leq \text{lowCut} < \text{highCut} \leq 1$), and then scales them by the *range* value.

If we denote by x_k the input values (which are equal to $pSrc[k]$ for not-in-place functions and to $pSrcDst[k]$ for in-place functions), then the output values y_k (written to $pDst[k]$ for not-in-place functions and to $pSrcDst[k]$ for in-place functions) are calculated according to the following formula:

$$y_k = \max \left(0, \min \left(range, \frac{\frac{x_k - x_{min}}{x_{max} - x_{min}} - lowCut}{highCut - lowCut} \cdot range \right) \right), k = 0, \dots, len - 1,$$

where

$x_{min} = \min_{i=0, \dots, len-1} x_i$, $x_{max} = \max_{i=0, \dots, len-1} x_i$ for `ippsNormalizeInRange` function
and

$x_{min} = valMin$, $x_{max} = valMax$ for `ippsNormalizeInRangeMinMax` function.

This function can be used to form spectrogram data.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates an error when the condition $0 \leq lowCut < highCut \leq 1$ is not met, or <i>range</i> is less than 0, or <i>valMin</i> is greater than <i>valMax</i> .
<code>ippStsInvZero</code>	Indicates a warning that $x_{min} = x_{max}$. All elements of the output vector are set to 0.

MeanVarAcc

Accumulates the estimates for the mean and variance re-estimation.

Syntax

```

IppStatus ippsMeanVarAcc_32f(Ipp32f const* pSrc, Ipp32f const* pSrcMean,
                             Ipp32f* pDstMeanAcc, Ipp32f* pDstVarAcc, int len, Ipp32f val);
IppStatus ippsMeanVarAcc_64f(Ipp64f const* pSrc, Ipp64f const* pSrcMean,
                             Ipp64f* pDstMeanAcc, Ipp64f* pDstVarAcc, int len, Ipp64f val);

```

Parameters

<i>pSrc</i>	Pointer to the observation vector [<i>len</i>].
<i>pSrcMean</i>	Pointer to the old mean vector [<i>len</i>].
<i>pDstMeanAcc</i>	Pointer to the mean accumulator [<i>len</i>].
<i>pDstVarAcc</i>	Pointer to the variance accumulator [<i>len</i>].
<i>len</i>	Length of the observation vector.
<i>val</i>	Constant value in the re-estimation.

Description

The function `ippsMeanVarAcc` is declared in the `ippsr.h` file. This function accumulates the estimates for the mean and variance re-estimation in the forward-backward algorithm. The calculation is as follows:

$$pDstMeanAcc[i] = pDstMeanAcc[i] + val \cdot (pSrc[i] - pSrcMean[i]) ,$$

$$pDstVarAcc[i] = pDstVarAcc[i] + val \cdot (pSrc[i] - pSrcMean[i])^2 ,$$

$$0 \leq i < len .$$

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> , <code>pSrcMean</code> , <code>pDstMeanAcc</code> , or <code>pDstVarAcc</code> pointer is NULL.
<code>ippsStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

GaussianDist

Calculates the distance between two Gaussians.

Syntax

```
IppStatus ippsGaussianDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
    Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f wgt1,
    Ipp32f det1, Ipp32f wgt2, Ipp32f det2);

IppStatus ippsGaussianDist_64f(const Ipp64f* pMean1, const Ipp64f* pVar1, const
    Ipp64f* pMean2, const Ipp64f* pVar2, int len, Ipp64f* pResult, Ipp64f wgt1,
    Ipp64f det1, Ipp64f wgt2, Ipp64f det2);
```

Parameters

<code>pMean1</code>	Pointer to the mean vector of the first Gaussian [<code>len</code>].
<code>pVar1</code>	Pointer to the variance vector of the first Gaussian [<code>len</code>].
<code>pMean2</code>	Pointer to the mean vector of the second Gaussian [<code>len</code>].
<code>pVar2</code>	Pointer to the variance vector of the second Gaussian [<code>len</code>].

<code>len</code>	Length of the mean and variance vectors.
<code>pResult</code>	Pointer to the distance value.
<code>wgt1</code>	Weight of the first Gaussian.
<code>det1</code>	Determinant of the first Gaussian (in the logarithmic representation).
<code>wgt2</code>	Weight of the second Gaussian.
<code>det2</code>	Determinant of the second Gaussian (in the logarithmic representation).

Description

The function `ippsGaussianDist` is declared in the `ippsr.h` file. This function calculates the distance between two Gaussians as follows:

$$pResult[0] = (wgt1 \cdot det1) + (wgt2 \cdot det2) - (wgt1 + wgt2) \cdot (len \cdot \ln(2\pi) - \ln(V)) ,$$

where

$$V = \prod_{i=0}^{len-1} \frac{W_1[i] + W_2[i] - (wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i])^2}{wgt1 + wgt2} ,$$

and

$$W_1[i] = wgt1 \cdot (pVar1[i] + pMean1[i]^2) ,$$

$$W_2[i] = wgt2 \cdot (pVar2[i] + pMean2[i]^2)$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMean1</code> , <code>pMean2</code> , <code>pVar1</code> , <code>pVar2</code> , or <code>pResult</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>-Inf</code> .
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>NaN</code> .

GaussianSplit

Splits a single Gaussian component into two with the same variance.

Syntax

```
IppStatus ippsGaussianSplit_32f(Ipp32f* pMean1, Ipp32f* pVar1, Ipp32f* pMean2,
                                Ipp32f* pVar2, int len, Ipp32f val);
IppStatus ippsGaussianSplit_64f(Ipp64f* pMean1, Ipp64f* pVar1, Ipp64f* pMean2,
                                Ipp64f* pVar2, int len, Ipp64f val);
```

Parameters

<i>pMean1</i>	Pointer to the input Gaussian mean vector, also the mean vector of the first Gaussian after splitting [<i>len</i>].
<i>pVar1</i>	Pointer to the input Gaussian variance vector, also the variance vector of the first Gaussian after splitting [<i>len</i>].
<i>pMean2</i>	Pointer to the mean vector of the second Gaussian after splitting [<i>len</i>].
<i>pVar2</i>	Pointer to the variance vector of the second Gaussian after splitting [<i>len</i>].
<i>len</i>	Length of the mean and variance vectors.
<i>val</i>	Variance perturbation value.

Description

The function `ippsGaussianSplit` is declared in the `ippsr.h` file. This function splits the Gaussian component into two Gaussian mixture components as follows:

$$pMean1[i] = pMean1[i] + val \cdot \sqrt{pVar1[i]},$$

$$pMean2[i] = pMean1[i] - val \cdot \sqrt{pVar1[i]},$$

$$pVar2[i] = pVar1[i],$$

$$0 \leq i < len.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMean1</code> , <code>pMean2</code> , <code>pVar1</code> , or <code>pVar2</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

GaussianMerge

Merges two Gaussian probability distribution functions.

Syntax

```
IppStatus ippsGaussianMerge_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
    const Ipp32f* pMean2, const Ipp32f* pVar2, Ipp32f* pDstMean, Ipp32f*
    pDstVar, int len, Ipp32f* pDstDet, Ipp32f wgt1, Ipp32f wgt2);

IppStatus ippsGaussianMerge_64f(const Ipp64f* pMean1, const Ipp64f* pVar1,
    const Ipp64f* pMean2, const Ipp64f* pVar2, Ipp64f* pDstMean, Ipp64f*
    pDstVar, int len, Ipp64f* pDstDet, Ipp64f wgt1, Ipp64f wgt2);
```

Parameters

<code>pMean1</code>	Pointer to the mean vector of the first Gaussian [<code>len</code>].
<code>pVar1</code>	Pointer to the variance vector of the first Gaussian [<code>len</code>].
<code>pMean2</code>	Pointer to the mean vector of the second Gaussian [<code>len</code>].
<code>pVar2</code>	Pointer to the variance vector of the second Gaussian [<code>len</code>].
<code>len</code>	Length of the mean and variance vectors.
<code>pDstMean</code>	Pointer to the mean vector of the merged Gaussian [<code>len</code>].
<code>pDstVar</code>	Pointer to the variance vector of the merged Gaussian [<code>len</code>].
<code>pDstDet</code>	Pointer to the determinant of the merged Gaussian.
<code>wgt1</code>	Weight of the first Gaussian.
<code>wgt2</code>	Weight of the second Gaussian.

Description

The function `ippsGaussianMerge` is declared in the `ippsr.h` file. This function merges two Gaussian probability distribution functions as follows:

$$pDstMean[i] = \frac{wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i]}{wgt1 + wgt2} ,$$

$$pDstVar[i] = \frac{wgt1^2 \cdot (pVar1[i] + pMean1[i]^2) + wgt2^2 \cdot (pVar2[i] + pMean2[i]^2)}{wgt1 + wgt2} ,$$

$$pDstDet[0] = len \cdot \ln(2\pi) - \sum_{i=0}^{len-1} \ln pDstVar[i] ,$$

$0 \leq i < len$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMean1</code> , <code>pMean2</code> , <code>pVar1</code> , <code>pVar2</code> , <code>pDstMean</code> , <code>pDstVar</code> , or <code>pDstDet</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>−Inf</code> .
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>NaN</code> .

Entropy

Calculates entropy of the input vector.

Syntax

```

IppStatus ippEntropy_32f(const Ipp32f* pSrc, int len, Ipp32f* pResult);
IppStatus ippEntropy_16s32s_Sfs(const Ipp16s* pSrc, int srcShiftVal, int len,
    Ipp32s* pResult, int scaleFactor);

```


Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len</i>].
<i>pResult</i>	Pointer to the destination entropy value.
<i>len</i>	Length of the input vector.
<i>srcShiftVal</i>	Refer to “Integer Scaling” in Chapter 2.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsEntropy` is declared in the `ippsr.h` file. This function calculates entropy of the input vector as given by:

$$pResult[0] = \sum_{i=0}^{len-1} pSrc[i] \cdot \log_2 pSrc[i]$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pResult</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnNegArg</code>	Indicates a warning that some input vector elements are less than 0. Operation execution is not aborted. For floating-point operations the destination value is set to NaN, and for integer operations it is set to 0.

SinC

Calculates sine divided by its argument.

Syntax

```
IppStatus ippsSinc_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinc_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinc_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinc_32f_I(Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsSinc_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len</i>].
<i>pSrcDst</i>	Pointer to the input and destination vector [<i>len</i>].
<i>pDst</i>	Pointer to the destination vector [<i>len</i>].
<i>len</i>	Length of the input and output vectors.

Description

The function `ippsSinc` is declared in the `ippsr.h` file. This function calculates destination vector elements according to the formula:

$$pDst[k] = \frac{\sin(pSrc[k])}{pSrc[k]}, k = 0, \dots, len - 1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pSrcDst</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

ExpNegSqr

Calculates exponential of the squared argument taken with the inverted sign.

Syntax

```
IppStatus ippsExpNegSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExpNegSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExpNegSqr_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExpNegSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsExpNegSqr_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len</i>].
-------------	---

<i>pSrcDst</i>	Pointer to the input and destination vector [<i>len</i>].
<i>pDst</i>	Pointer to the destination vector [<i>len</i>].
<i>len</i>	Length of the input and output vectors.

Description

The function `ippsExpNegSqr` is declared in the `ippsr.h` file. This function calculates destination vector elements according to the following formula:

$$pDst[k] = \exp(-pSrc[k]^2), k = 0, \dots, len - 1.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pSrcDst</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

BhatDist

Calculates the Bhattacharia distance between two Gaussians.

Syntax

```

IppStatus ippsBhatDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
    Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult);
IppStatus ippsBhatDist_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
    Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult);
IppStatus ippsBhatDistSLog_32f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
    Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f
    sumLog1, Ipp32f sumLog2);
IppStatus ippsBhatDistSLog_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1,
    const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult, Ipp32f
    sumLog1, Ipp32f sumLog2);

```

Parameters

<i>pMean1</i>	Pointer to the first mean vector [<i>len</i>].
---------------	--

<i>pVar1</i>	Pointer to the first variance vector [<i>len</i>].
<i>pMean2</i>	Pointer to the second mean vector [<i>len</i>].
<i>pVar2</i>	Pointer to the second variance vector [<i>len</i>].
<i>pResult</i>	Pointer to the result.
<i>len</i>	Length of the input mean and variance vectors.
<i>sumLog1</i>	Sum of the first Gaussian variance in the logarithmic representation.
<i>sumLog2</i>	Sum of the second Gaussian variance in the logarithmic representation.

Description

The functions `ippsBhatDist` and `ippsBhatDistSLog` are declared in the `ippsr.h` file.

The function `ippsBhatDist` calculates the Bhattacharia distance between two Gaussians as follows:

$$pResult[0] = \frac{1}{4} \sum_{i=0}^{len-1} \frac{(pMean1[i] - pMean2[i])^2}{pVar1[i] + pVar2[i]} +$$

$$+ \frac{1}{2} \sum_{i=0}^{len-1} \left(\ln\left(\frac{pVar1[i] + pVar2[i]}{2}\right) - \frac{\ln(pVar1[i]) + \ln(pVar2[i])}{2} \right)$$

The function `ippsBhatDistSLog` calculates the Bhattacharia distance between two Gaussians as follows:

$$pResult[0] = \frac{1}{4} \sum_{i=0}^{len-1} \frac{(pMean1[i] - pMean2[i])^2}{pVar1[i] + pVar2[i]} +$$

$$+ \frac{1}{2} \sum_{i=0}^{len-1} \ln\left(\frac{pVar1[i] + pVar2[i]}{2}\right) - \frac{sumLog1 + sumLog2}{4}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pMean1</code> , <code>pVar1</code> , <code>pMean2</code> , <code>pVar2</code> , or <code>pResult</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

<code>ippStsLnZeroArg</code>	Indicates a warning that a zero value was detected in the input vector. The execution is not aborted. The result value is set to <code>-Inf</code> if there is no negative element in the vector.
<code>ippStsLnNegArg</code>	Indicates a warning that negative values were detected in the input vector. The execution is not aborted. The result value is set to <code>NaN</code> .

UpdateMean

Updates the mean vector in the EM training algorithm.

Syntax

```
IppStatus ippUpdateMean_32f(const Ipp32f* pMeanAcc, Ipp32f* pMean, int len,
                             Ipp32f meanOcc);
IppStatus ippUpdateMean_64f(const Ipp64f* pMeanAcc, Ipp64f* pMean, int len,
                             Ipp64f meanOcc);
```

Parameters

<code>pMeanAcc</code>	Pointer to the mean accumulator [<code>len</code>].
<code>pMean</code>	Pointer to the mean vector [<code>len</code>].
<code>len</code>	Length of the mean vector.
<code>meanOcc</code>	Occupation sum of the Gaussian mixture.

Description

The function `ippUpdateMean` is declared in the `ippsr.h` file. This function calculates the updated mean vector in the EM (Expectation-Maximization) training algorithm as follows:

$$pMean[i] = pMean[i] + \frac{pMeanAcc[i]}{meanOcc}, 0 \leq i < len.$$

Note that if $meanOcc \leq 0$, the mean vector `pMean` is not updated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pMean</code> or <code>pMeanAcc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

<code>ippStsZeroOcc</code>	Indicates a warning that <i>meanOcc</i> is equal to 0.
<code>ippStsNegOccErr</code>	Indicates an error when <i>meanOcc</i> is less than 0.

UpdateVar

Updates the variance vector in the EM training algorithm.

Syntax

```
IppStatus ippUpdateVar_32f(const Ipp32f* pMeanAcc, const Ipp32f* pVarAcc,
    const Ipp32f* pVarFloor, Ipp32f* pVar, int len, Ipp32f meanOcc, Ipp32f
    varOcc);

IppStatus ippUpdateVar_64f(const Ipp64f* pMeanAcc, const Ipp64f* pVarAcc,
    const Ipp64f* pVarFloor, Ipp64f* pVar, int len, Ipp64f meanOcc, Ipp64f
    varOcc);
```

Parameters

<i>pMeanAcc</i>	Pointer to the mean accumulator [<i>len</i>].
<i>pVarAcc</i>	Pointer to the variance accumulator [<i>len</i>].
<i>pVarFloor</i>	Pointer to the variance floor vector [<i>len</i>].
<i>pVar</i>	Pointer to the variance vector [<i>len</i>].
<i>len</i>	Length of the variance.
<i>meanOcc</i>	Occupation sum of the Gaussian mixture.
<i>varOcc</i>	Square occupation sum of the variance mixture.

Description

The function `ippUpdateVar` is declared in the `ippsr.h` file. This function calculates the updated variance vector in the EM algorithm. The covariance matrix is assumed to be diagonal. The accumulators are calculated from the training data. The update equation is as follows:

$$pVar[i] = \max \left[varFloor[i], \frac{varAcc[i]}{varOcc} - \left(\frac{meanAcc[i]}{meanOcc} \right)^2 \right], 0 \leq i < len.$$

Note that if $meanOcc \leq 0$ or $varOcc \leq 0$, the variance vector *pVar* is not updated.

Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when <i>pMeanAcc</i> , <i>pVarAcc</i> , <i>pVarFloor</i> , or <i>pVar</i> pointer is NULL.
<i>ippStsSizeErr</i>	Indicates an error when <i>len</i> is less than or equal to 0.
<i>ippStsZeroOcc</i>	Indicates a warning that <i>meanOcc</i> or <i>varOcc</i> is equal to 0.
<i>ippStsNegOccErr</i>	Indicates an error when <i>meanOcc</i> and <i>varOcc</i> are less than 0.
<i>ippStsResFloor</i>	Indicates a warning that all variances are floored.

UpdateWeight

Updates the weight values of Gaussian mixtures in the EM training algorithm.

Syntax

```
IppStatus ippsUpdateWeight_32f(const Ipp32f* pWgtAcc, Ipp32f* pWgt, int len,
                               Ipp32f* pWgtSum, Ipp32f wgtOcc, Ipp32f wgtThresh);
IppStatus ippsUpdateWeight_64f(const Ipp64f* pWgtAcc, Ipp64f* pWgt, int len,
                               Ipp64f* pWgtSum, Ipp64f wgtOcc, Ipp64f wgtThresh);
```

Parameters

<i>pWgtAcc</i>	Pointer to the weight accumulator [<i>len</i>].
<i>pWgt</i>	Pointer to the weight vector [<i>len</i>].
<i>len</i>	Number of Gaussian mixture components.
<i>pWgtSum</i>	Pointer to the output sum of weight values.
<i>wgtOcc</i>	Nominator of the weight update equation.
<i>wgtThresh</i>	Threshold for the weight values.

Description

The function `ippsUpdateWeight` is declared in the `ippsr.h` file. This function calculates the updated weight values for a Gaussian mixture. The accumulators are calculated from the training data. The update equation is as follows:

$$pWgt[i] = \max\left(\frac{pWgtAcc[i]}{wgtOcc}, wgtThresh\right), 0 \leq i < len$$

$$pWgtSum[0] = \sum_{i=0}^{len-1} pWgt[i].$$

Note that if $wgtOcc \leq 0$, the weight vector $pWgt$ is not updated.

This function can also be used to update the HMM transition matrix.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when $pWgt$, $pWgtAcc$, or $pWgtSum$ pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when len is less than or equal to 0.
<code>ippStsZeroOcc</code>	Indicates a warning that $wgtOcc$ is equal to 0.
<code>ippStsNegOccErr</code>	Indicates an error when $wgtOcc$ is less than 0.
<code>ippStsResFloor</code>	Indicates a warning that all weights are floored.

UpdateGConst

Updates the fixed constant in the Gaussian output probability density function.

Syntax

```
IppStatus ippsUpdateGConst_32f(const Ipp32f* pVar, int len, Ipp32f* pDet);
IppStatus ippsUpdateGConst_64f(const Ipp64f* pVar, int len, Ipp64f* pDet);
IppStatus ippsUpdateGConst_DirectVar_32f(const Ipp32f* pVar, int len,
    Ipp32f* pDet);
```



```
IppStatus ippsUpdateGConst_DirectVar_64f(const Ipp64f* pVar, int len,
    Ipp64f* pDet);
```

Parameters

<i>pVar</i>	Pointer to the variance vector [<i>len</i>].
<i>len</i>	Dimension of the variance vector.
<i>pDet</i>	Pointer to the result value.

Description

The function `ippsUpdateGConst` is declared in the `ippsr.h` file. This function calculates the fixed variance constant.

For functions without the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be inverse diagonal:

$$pDet[0] = len \cdot \ln 2\pi - \sum_{i=0}^{len-1} \ln(pVar[i])$$

For functions with the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be diagonal:

$$pDet[0] = len \cdot \ln 2\pi + \sum_{i=0}^{len-1} \ln(pVar[i])$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pVar</i> or <i>pDet</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The result value is set to <code>-Inf</code> if there are no negative elements in the vector.
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements. Operation execution is not aborted. The result value is set to <code>NaN</code> .

OutProbPreCalc

Pre-calculates the part of Gaussian mixture output probability that is irrelevant to observation vectors.

Syntax

```
IppStatus ippsOutProbPreCalc_32s(const Ipp32s* pWeight, const Ipp32s* pSrc,
    Ipp32s* pDst, int len);
IppStatus ippsOutProbPreCalc_32s_I(const Ipp32s* pWeight, Ipp32s* pSrcDst,
    int len);
IppStatus ippsOutProbPreCalc_32f(const Ipp32f* pWeight, const Ipp32f* pSrc,
    Ipp32f* pDst, int len);
IppStatus ippsOutProbPreCalc_64f(const Ipp64f* pWeight, const Ipp64f* pSrc,
    Ipp64f* pDst, int len);
IppStatus ippsOutProbPreCalc_32f_I(const Ipp32f* pWeight, Ipp32f* pSrcDst,
    int len);
IppStatus ippsOutProbPreCalc_64f_I(const Ipp64f* pWeight, Ipp64f* pSrcDst,
    int len);
```

Parameters

<i>pWeight</i>	Pointer to the Gaussian mixture weight vector [<i>len</i>].
<i>pSrc</i>	Pointer to the input vector calculated from ippsUpdateGConst function.
<i>pSrcDst</i>	Pointer to the input and output vector calculated from ippsUpdateGConst function.
<i>pDst</i>	Pointer to the result vector [<i>len</i>].
<i>len</i>	Number of mixtures in the HMM state.

Description

The function `ippsOutProbPreCalc` is declared in the `ippsr.h` file. This function pre-calculates the part of the Gaussian mixture output probability that is irrelevant to the observation vectors.

For the function `ippsOutProbPreCalc`,

$$pDst[i] = pWeight[i] - 0.5 \cdot pSrc[i], 0 \leq i < len.$$

For the function `ippsOutProbPreCalc_I`,

$$pSrcDst[i] = pWeight[i] - 0.5 \cdot pSrcDst[i], 0 \leq i < len.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pWeight</i> , <i>pDet</i> , or <i>pVal</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when the <i>len</i> is less than or equal to zero.

DcsClustLAccumulate

Updates the accumulators for calculating the state-cluster likelihood in the decision-tree clustering algorithm.

Syntax

```

IppStatus ippsDcsClustLAccumulate_32f(const Ipp32f* pMean, const Ipp32f* pVar,
    Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);
IppStatus ippsDcsClustLAccumulate_64f(const Ipp64f* pMean, const Ipp64f* pVar,
    Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);
IppStatus ippsDcsClustLAccumulate_DirectVar_32f(const Ipp32f* pMean, const
    Ipp32f* pVar, Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);
IppStatus ippsDcsClustLAccumulate_DirectVar_64f(const Ipp64f* pMean, const
    Ipp64f* pVar, Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);

```

Parameters

<i>pMean</i>	Pointer to the mean vector of an HMM state in the cluster [<i>len</i>].
<i>pVar</i>	Pointer to the variance vector of an HMM state in the cluster [<i>len</i>].
<i>pDstSum</i>	Pointer to the summation part of the accumulator [<i>len</i>].
<i>pDstSqr</i>	Pointer to the square sum part of the accumulator [<i>len</i>].
<i>len</i>	Length of the mean and variance vectors.
<i>occ</i>	Occupation counts of the HMM state

Description

The function `ippsDcsClustLAccumulate` is declared in the `ippsr.h` file. This function updates the accumulators in the decision-tree clustering algorithm. The accumulators are used to calculate the likelihood of an HMM state cluster.

The accumulation equations are as follows:

For functions without the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be inverse diagonal, and

$$pDstSum[i] = pMean[i] \cdot occ ,$$

$$pDstSqr[i] = \left[\frac{1}{pVar[i]} + (pMean[i])^2 \right] \cdot occ , \text{ for } 0 \leq i < len.$$

For functions with the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be diagonal, and

$$pDstSum[i] = pMean[i] \cdot occ ,$$

$$pDstSqr[i] = [pVar[i] + (pMean[i])^2] \cdot occ , \text{ for } 0 \leq i < len.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pMean</code> , <code>pVar</code> , <code>pDstSum</code> , or <code>pDstSqr</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

DcsClustLCompute

Calculates the likelihood of an HMM state cluster in the decision-tree state-clustering algorithm.

Syntax

```
IppStatus ippsDcsClustLCompute_64f(const Ipp64f* pSrcSum, const Ipp64f*
    pSrcSqr, int len, Ipp64f* pDst, Ipp64f occ);

IppStatus ippsDcsClustLCompute_32f64f(const Ipp32f* pSrcSum, const Ipp32f*
    pSrcSqr, int len, Ipp64f* pDst, Ipp32f occ);
```

Parameters

<i>pSrcSum</i>	Pointer to the summation part of the accumulator [<i>len</i>].
<i>pSrcSqr</i>	Pointer to the square sum part of the accumulator [<i>len</i>].
<i>len</i>	Length of the <i>pSrcSum</i> and <i>pSrcSqr</i> vectors.
<i>pDst</i>	Pointer to the result likelihood value.
<i>occ</i>	Occupation sum of the HMM state cluster.

Description

The function `ippsDcsClustLCompute` is declared in the `ippsr.h` file. This function calculates the likelihood of an HMM state cluster, according to the accumulators computed by `ippsDcsClustLAccumulate` function. The likelihood value is used to determine the splitting of a decision tree node in the decision-tree state-clustering algorithm. The calculation is as follows:

$$pDst[0] = -\frac{1}{2}occ \cdot \left\{ len \cdot [1 + \ln 2\pi - 2\ln(occ)] + \sum_{i=0}^{len-1} \ln[pSrcSqr[i] \cdot occ - (pSrcSum[i])^2] \right\}$$

Note that if $occ = 0$, $pDst[0]$ is set to `IPPLGZERO`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcSum</i> , <i>pSrcSqr</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0 or <i>occ</i> is less than or equal to 0.
<code>ippStsZeroOcc</code>	Indicates a warning that <i>occ</i> is equal to 0.
<code>ippStsNegOccErr</code>	Indicates an error when <i>occ</i> is less than 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The result value is set to <code>-Inf</code> if there are no negative elements in the vector.
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements. Operation execution is not aborted. The result value is set to <code>NaN</code> .

Model Adaptation

This section describes functions that can be used to adapt the acoustic and language models. The adaptation algorithms adjust the parameters of existing models to match the characteristics set by the users, given a few learning samples.

AddMulColumn

Adds a weighted matrix column to the other column.

Syntax

```
IppStatus ippsAddMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height, int
    col1, int col2, int row1, const Ipp64f val);
```

Parameters

<i>mSrcDst</i>	Pointer to the source and destination matrix [<i>height</i>][<i>width</i>].
<i>width</i>	Number of columns in the matrix <i>mSrcDst</i> .
<i>height</i>	Number of rows in the matrix <i>mSrcDst</i> .
<i>col1</i>	Column number of the first operand.
<i>col2</i>	Column number of the second operand.
<i>row1</i>	Starting row number.
<i>val</i>	Weight factor.

Description

The function `AddMulColumn` is declared in the `ippsr.h` file. This function adds a matrix column weighted by *val* to the other matrix column. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2] + mSrcDst[i][col1] \cdot val,$$

for $row1 \leq i < height$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when <i>mSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>width</i> , <i>col1</i> , <i>col2</i> , or <i>row1</i> is less than or equal to 0; or <i>col1</i> or <i>col2</i> is greater than or equal to <i>width</i> ; or <i>row1</i> is greater than or equal to <i>height</i> .

AddMulRow

Adds a weighted vector to the other vector.

Syntax

```
ippsAddMulRow_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
                  const Ipp64f val);
```

<i>pSrc</i>	Pointer to the source vector [<i>len</i>].
<i>pSrcDst</i>	Pointer to the source and destination vector [<i>len</i>].
<i>len</i>	Length of the source and destination vectors.
<i>val</i>	Weight factor.

Description

The function `AddMulRow` is declared in the `ippsr.h` file. This function adds a vector weighted by *val* to the other vector. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$pSrcDst[i] = pSrcDst[i] + pSrc[i] \cdot val, \quad 0 \leq i < len.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

QRTransColumn

Performs the QR transformation.

Syntax

```
IppStatus ippsQRTransColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
    int col1, int col2, const Ipp64f val1, const Ipp64f val2);
```

Parameters

<i>mSrcDst</i>	Pointer to the source matrix and destination matrix [<i>height</i>][<i>width</i>].
<i>width</i>	Number of columns in the matrix <i>mSrcDst</i> .
<i>height</i>	Number of rows in the matrix <i>mSrcDst</i> .
<i>col1</i>	First column number.
<i>col2</i>	Second column number.
<i>val1</i>	First weight factor.
<i>val2</i>	Second weight factor.

Description

The `ippsQRTransColumn` is declared in the `ippsr.h` file. This function performs the *QR* transform. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2] \cdot val1 + mSrcDst[i][col1] \cdot val2,$$

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val1 + mSrcDst[i][col2] \cdot val2,$$

for $0 \leq i < height$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>mSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>width</i> , <i>col1</i> , or <i>col2</i> is less than or equal to 0; or <i>col1</i> or <i>col2</i> is greater than or equal to <i>width</i> ; or <i>col2</i> is greater than or equal to <i>height</i> .

DotProdColumn

Calculates the dot product of two matrix columns.

Syntax

```
IppStatus ippsDotProdColumn_64f_D2L(const Ipp64f** mSrc, int width,  
    int height, Ipp64f* pSum, int col1, int col2, int row1);
```

Parameters

<i>mSrc</i>	Pointer to the source matrix [<i>height</i>][<i>width</i>].
<i>width</i>	Number of columns in the matrix <i>mSrc</i> .
<i>height</i>	Number of rows in the matrix <i>mSrc</i> .
<i>pSum</i>	Pointer to the computed sum.
<i>col1</i>	First column number.
<i>col2</i>	Second column number.
<i>row1</i>	First row number.

Description

The function `ippsDotProdColumn` is declared in the `ippsr.h` file. This function calculates the dot product of two matrix columns. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$pSum[0] = \sum_{i=row1}^{height-1} mSrc[i][col1] \cdot mSrc[i][col2]$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>mSrc</i> or <i>pSum</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>width</i> , <i>col1</i> , or <i>row1</i> is less than or equal to 0; or <i>row1</i> is greater than or equal to <i>height</i> ; or <i>col1</i> is greater than or equal to <i>width</i> .

MulColumn

Multiplies a matrix column by a value.

Syntax

```
IppStatus ippsMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
    int col1, int row1, const Ipp64f val);
```

Parameters

<i>mSrcDst</i>	Pointer to the source and destination matrix [<i>height</i>][<i>width</i>].
<i>width</i>	Number of columns in the matrix <i>mSrcDst</i> .
<i>height</i>	Number of rows in the matrix <i>mSrcDst</i> .
<i>col1</i>	First column number.
<i>row1</i>	First row number.
<i>val</i>	Weight factor.

Description

The function `ippsMulColumn` is declared in the `ippsr.h` file. This function multiplies a column of the *mSrcDst* matrix by *val*. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val, \text{ row1} \leq i < height.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>mSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>width</i> , <i>col1</i> , or <i>row1</i> is less than or equal to 0; or <i>row1</i> is greater than or equal to <i>height</i> ; or <i>col1</i> is greater than or equal to <i>width</i> .

SumColumnAbs

Calculates the absolute sum of matrix column elements.

Syntax

```
IppStatus ippsSumColumnAbs_64f_D2L(const Ipp64f** mSrc, int width,
    int height, Ipp64f* pSum, int coll, int row1);
```

Parameters

<i>mSrc</i>	Pointer to the source matrix [<i>height</i>][<i>width</i>].
<i>width</i>	Number of columns in the matrix <i>mSrc</i> .
<i>height</i>	Number of rows in the matrix <i>mSrc</i> .
<i>pSum</i>	Pointer to the computed sum.
<i>coll</i>	First column number.
<i>row1</i>	First row number.

Description

The function `ippsSumColumnAbs` is declared in the `ippsr.h` file. This function calculates the absolute sum of the matrix column elements. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$pSum[0] = \sum_{i=row1}^{height-1} |mSrc[i][coll]|$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>mSrc</i> or <i>pSum</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>height</i> , <i>coll</i> , or <i>row1</i> is less than or equal to 0; or <i>row1</i> is greater than or equal to <i>height</i> ; or <i>coll</i> is greater than or equal to <i>width</i> .

SumColumnSqr

Calculates the square sums of weighted matrix column elements.

Syntax

```
IppStatus ippsSumColumnSqr_64f_D2L(Ipp64f** mSrcDst, int width, int height,
    Ipp64f* pSum, int coll, int row1, const Ipp64f val);
```

Parameters

<i>mSrcDst</i>	Pointer to the source and destination matrix [<i>height</i>][<i>width</i>].
<i>width</i>	Number of columns in the matrix <i>mSrcDst</i> .
<i>height</i>	Number of rows in the matrix <i>mSrcDst</i> .
<i>pSum</i>	Pointer to the value of the computed sum.
<i>coll</i>	First column number.
<i>row1</i>	Second row number.
<i>val</i>	Weight factor.

Description

The function `ippsSumColumnSqr` is declared in the `ippsr.h` file. This function multiplies the columns of the matrix *mSrcDst* by *val* and calculates the square sums of the column elements. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][coll] = mSrcDst[i][coll] \cdot val, \quad 0 \leq i < height$$

$$pSum[0] = \sum_{i=row1}^{height-1} (mSrcDst[i][coll])^2$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>mSrcDst</i> or <i>pSum</i> pointer is NULL.

<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>height</i> , <i>coll</i> , or <i>row1</i> is less than or equal to 0; or <i>row1</i> is greater than or equal to <i>height</i> ; or <i>coll</i> is greater than or equal to <i>width</i> .
----------------------------	--

SumRowAbs

Calculates the absolute sum of the vector elements.

Syntax

```
IppStatus ippSumRowAbs_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
```

Parameters

<i>pSrc</i>	Pointer to the source vector [<i>len</i>].
<i>pSum</i>	Pointer to the value of the computed sum.
<i>len</i>	Length of the source vector <i>pSrc</i> .

Description

The function `ippSumRowAbs` is declared in the `ippsr.h` file. This function calculates the absolute sum of the vector elements as follows:

$$pSum[0] = \sum_{i=0}^{len-1} |pSrc[i]|$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pSum</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

SumRowSqr

Calculates the square sum of weighted vector elements.

Syntax

```
IppStatus ippsSumRowSqr_64f(Ipp64f* pSrcDst, int len, Ipp64f* pSum,
    const Ipp64f val);
```

Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector [<i>len</i>].
<i>len</i>	Length of the source vector <i>pSrcDst</i> .
<i>pSum</i>	Pointer to the value of the computed sum.
<i>val</i>	Weight factor.

Description

The function `ippsSumRowSqr` is declared in the `ippsr.h` file. This function multiplies the vector *pSrcDst* by *val* and calculates the square sum of the vector elements as follows:

$pSrcDst[i] = pSrcDst[i] \cdot val, \quad 0 \leq i < len$

$$pSum[0] = \sum_{i=0}^{len-1} (pSrcDst[i])^2.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcDst</i> or <i>pSum</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

SVD, SVDSort

*Performs single value decomposition
on a matrix.*

Syntax

```
IppStatus ippsSVD_64f_D2(const Ipp64f* pSrcA, Ipp64f* pDstU, int height,
    Ipp64f* pDstW, Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVD_64f_D2L(const Ipp64f** mSrcA, Ipp64f** mDstU, int height,
    Ipp64f* pDstW, Ipp64f** mDstV, int width, int nIter);
IppStatus ippsSVD_64f_D2_I(Ipp64f* pSrcDstA, int height, Ipp64f* pDstW, Ipp64f*
    pDstV, int width, int step, int nIter);
IppStatus ippsSVD_64f_D2L_I(Ipp64f** mSrcDstA, int height, Ipp64f* pDstW,
    Ipp64f** mDstV, int width, int nIter);
IppStatus ippsSVDSort_64f_D2(const Ipp64f* pSrcA, Ipp64f* pDstU, int height,
    Ipp64f* pDstW, Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVDSort_64f_D2L(const Ipp64f** mSrcA, Ipp64f** mDstU, int height,
    Ipp64f* pDstW, Ipp64f** mDstV, int width, int nIter);
IppStatus ippsSVDSort_64f_D2_I(Ipp64f* pSrcDstA, int height, Ipp64f* pDstW,
    Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVDSort_64f_D2L_I(Ipp64f** mSrcDstA, int height, Ipp64f* pDstW,
    Ipp64f** mDstV, int width, int nIter);
```

Parameters

<i>pSrcA</i>	Pointer to the input vector A [$height \times step$].
<i>pDstU</i>	Pointer to the output vector U [$height \times step$].
<i>pSrcDstA</i>	Pointer to the input matrix A and output matrix U [$height \times step$].
<i>pDstV</i>	Pointer to the output vector V [$width \times step$].
<i>mSrcA</i>	Pointer to the input matrix A [$height$][$width$].
<i>mDstU</i>	Pointer to the output matrix U [$height$][$width$].
<i>mSrcDstA</i>	Pointer to the input matrix A and output matrix U [$height$][$width$].
<i>pDstW</i>	Pointer to the output vector W [$width$].

<i>mDstV</i>	Pointer to the output matrix V [<i>width</i>][<i>width</i>].
<i>height</i>	Number of rows in the input matrix.
<i>width</i>	Number of columns in the input matrix.
<i>step</i>	Row step in the vector <i>pSrcA</i> , <i>pSrcDstA</i> , or <i>pDstV</i> .
<i>nIter</i>	Number of iterations for diagonalization.

Description

The functions `ippsSVD` and `ippsSVDSort` are declared in the `ippsr.h` file. This function performs Single Value Decomposition (SVD) on the input matrix A . The output matrices U , W , and V meet the following condition:

$$A = U \circ W \circ V^T,$$

where the matrix U is column-orthogonal, the matrix W is diagonal (stored as a vector), and the matrix V is orthogonal. V^T is the transpose of the matrix V .

The function `ippsSVDSort` transforms output matrices so that elements of the matrix W are sorted in descending order of their absolute values.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>width</i> , <i>step</i> , or <i>nIter</i> is less than or equal to 0, or <i>width</i> is greater than <i>step</i> .
<code>ippStsSVDConvErr</code>	Indicates an error when the SVD algorithm has not converged after <i>nIter</i> iterations.

WeightedSum

Calculates the weighted sums of two input vector elements.

Syntax

```
IppStatus ippsWeightedSum_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSum_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSum_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len, Ipp64f weight1, Ipp64f weight2);
IppStatus ippsWeightedSumHalf_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);
```

Parameters

<i>pSrc1</i>	Pointer to the first input vector [<i>len</i>].
<i>pSrc2</i>	Pointer to the second input vector [<i>len</i>].
<i>pDst</i>	Pointer to the output vector [<i>len</i>].
<i>len</i>	Length of the input and output vectors.
<i>weight1</i>	First weight value.
<i>weight2</i>	Second weight value.

Description

The functions `ippsWeightedSum` and `ippsWeightedSumHalf` are declared in the `ippsr.h` file. The function `ippsWeightedSum` calculates the weighted sum as follows:

$$pDst[i] = \frac{weight1 \cdot pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2} \quad , \quad i = 0, ..len-1.$$

The function `ippsWeightedSumHalf` calculates the weighted sum as given by:

$$pDst[i] = \frac{pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2}, \quad i = 0, \dots, len-1$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.						
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> , <code>pSrc2</code> , or <code>pDst</code> pointer is NULL.						
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.						
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value. The execution is not aborted. The value of the destination vector element for the floating-point operations is set as follows: <table data-bbox="617 659 1323 788"> <tr> <td>NaN</td><td>For zero-valued dividend vector element;</td></tr> <tr> <td>+Inf</td><td>For positive dividend vector element;</td></tr> <tr> <td>-Inf</td><td>For negative dividend vector element.</td></tr> </table>	NaN	For zero-valued dividend vector element;	+Inf	For positive dividend vector element;	-Inf	For negative dividend vector element.
NaN	For zero-valued dividend vector element;						
+Inf	For positive dividend vector element;						
-Inf	For negative dividend vector element.						

Vector Quantization

This section describes some functions for vector quantization and codebook operations. These functions are commonly used in acoustic and language model compressions.

FormVector

Constructs an output vector of multiple streams from codebook entries.

Syntax

```
IppStatus ippsFormVector_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc, const
    Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
    Ipp16s* pDst);

IppStatus ippsFormVector_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc, const
    Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
    Ipp16s* pDst);
```

```
IppStatus ippsFormVector_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc, const
    Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
    Ipp32f* pDst);
```

```
IppStatus ippsFormVector_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc, const
    Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
    Ipp32f* pDst);
```

```
IppStatus ippsFormVector_2i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc, const
    Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_2i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
    const Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_2i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc, const
    Ipp32s* pHeights, Ipp32f* pDst, int len);
```

```
IppStatus ippsFormVector_2i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
    const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

```
IppStatus ippsFormVector_4i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc, const
    Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_4i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
    const Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_4i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc, const
    Ipp32s* pHeights, Ipp32f* pDst, int len);
```

```
IppStatus ippsFormVector_4i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
    const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

Parameters

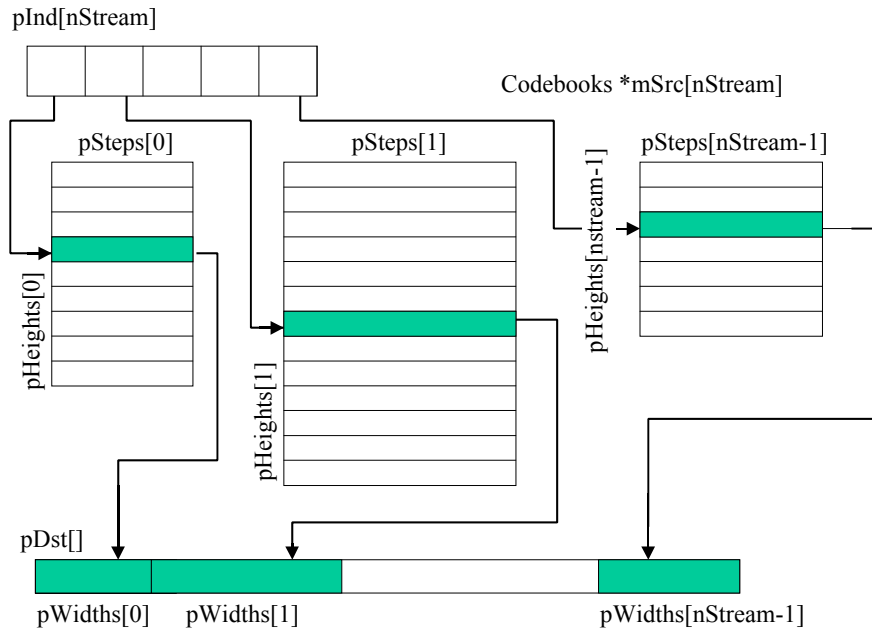
<i>pInd</i>	Pointer to the indexing vector [<i>nStream</i>].
<i>mSrc</i>	Pointer to the array of pointers to the codebooks [<i>nStream</i>].
<i>pHeights</i>	Pointer to the codebook lengths [<i>nStream</i>].
<i>pWidths</i>	Pointer to the stream lengths [<i>nStream</i>].
<i>pSteps</i>	Pointer to the codevector lengths [<i>nStream</i>].
<i>nStream</i>	Number of codebooks.
<i>pDst</i>	Pointer to the output vector.
<i>len</i>	Length of the output vector.

Description

The function `ippsFormVector` is declared in the `ippsr.h` file. This function constructs an output vector of multiple streams. Each stream, of size `pWeights[]`, is a codebook entry indexed by `pInd[]`. The codebooks are referenced by `mSrc[]`, and have `pHeights[]` number of codevectors, each of which is of size `pSteps[]`.

The following figure illustrates the layout:

Figure 8-5 Stream Layout for the `ippsFormVector` Function



The output vector is obtained as follows:

$$pDst \left[i + \sum_{j=0}^{k-1} pWidths[j] \right] = mSrc[k][pInd[k] \cdot pSteps[k] + i] ,$$

$k = 0, \dots, nStream - 1; i = 0, \dots, pWidths[k] - 1$.

The function `ippsFormVector_2i` simplifies the extraction process by posting the constraints of $pWidths[] = pSteps[] = 2$ and $nStream = len/2$.

Similarly, the function `ippsFormVector_4i` implies the constraints of $pWidths[] = pSteps[] = 4$ and $nStream = len/4$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when $pInd[k]$ is less than 0, or when $len, nStream, pWidths[k]$, or $pSteps[k]$ is less than or equal to 0; or when $pHeights[k]$ is less than or equal to $pInd[k]$.

CdbkGetSize

Calculates the size in bytes of the codebook.

Syntax

```
IppStatus ippsCdbkGetSize_16s(int width, int step, int height, int cdbkSize,
    IppCdbk_Hint hint, int* pSize);
```

Parameters

<i>width</i>	Length of the input vectors ($0 < width < 512$).
<i>step</i>	Row step in the source vector <code>pSrc</code> ($0 < step < 512$).
<i>height</i>	Number of rows in the source vector <code>pSrc</code> (currently only $height = cdbkSize$ is supported).
<i>cdbkSize</i>	Size of the codebook ($0 < cdbkSize \leq 8192$).
<i>hint</i>	Flag indicating format of codebook. See <code>ippsCdbkInit</code> for a complete description.
<i>pSize</i>	Pointer to the variable to contain the size in bytes of the codebook structure and associated storage.

Description

The function `ippsCdbkGetSize` is declared in the `ippsr.h` file. This function calculates the size in bytes of the codebook and additional information to be used for fast search.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>cdbkSize</i> is less than or equal to 0; or <i>cdbkSize</i> is greater than 8192; or <i>cdbkSize</i> is not equal to <i>height</i> ; or when <i>width</i> , <i>step</i> , or <i>height</i> is less than or equal to 0; or <i>width</i> is greater than <i>step</i> .
<code>ippStsCdbkFlagErr</code>	Indicates an error when the <i>hint</i> value is incorrect or not supported.



NOTE. The only *hint* value that is currently supported for this function is `IPP_CDBK_FULL`. The *height* parameter is not needed in this mode and its value will be ignored.

CdbkInit

Initializes the structure that contains the codebook.

Syntax

```
IppStatus ippsCdbkInit_L2_16s(IppsCdbkState_16s* pCdbk, const Ipp16s* pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
```

Parameters

<i>pSrc</i>	Pointer to the source vector with <i>height</i> * <i>step</i> entries.
<i>width</i>	Length of the input vectors ($0 < \textit{width} < 512$).
<i>step</i>	Row step in the source vector <i>pSrc</i> ($0 < \textit{step} < 512$).
<i>height</i>	Number of rows in the source vector <i>pSrc</i> ($0 < \textit{height} \leq \textit{cdbkSize}$).

<i>cdbkSize</i>	Size of the codebook ($0 < cdbkSize \leq 8192$).						
<i>hint</i>	One of the following values: <table> <tr> <td>IPP_CDBK_FULL</td><td>The source data are entries of a codebook, <i>height</i> should be greater or equal to <i>nCluster</i>. The nearest codebook entry is located through a full search.</td></tr> <tr> <td>IPP_CDBK_KMEANS_LONG</td><td>LBG algorithm with splitting of the most extensional cluster was used for the codebook building. The nearest codebook entry is located through a logarithmical search.</td></tr> <tr> <td>IPP_CDBK_KMEANS_NUM</td><td>LBG algorithm with splitting of the most numerous clusters was used for the codebook building. The nearest codebook entry is located through a logarithmical search.</td></tr> </table>	IPP_CDBK_FULL	The source data are entries of a codebook, <i>height</i> should be greater or equal to <i>nCluster</i> . The nearest codebook entry is located through a full search.	IPP_CDBK_KMEANS_LONG	LBG algorithm with splitting of the most extensional cluster was used for the codebook building. The nearest codebook entry is located through a logarithmical search.	IPP_CDBK_KMEANS_NUM	LBG algorithm with splitting of the most numerous clusters was used for the codebook building. The nearest codebook entry is located through a logarithmical search.
IPP_CDBK_FULL	The source data are entries of a codebook, <i>height</i> should be greater or equal to <i>nCluster</i> . The nearest codebook entry is located through a full search.						
IPP_CDBK_KMEANS_LONG	LBG algorithm with splitting of the most extensional cluster was used for the codebook building. The nearest codebook entry is located through a logarithmical search.						
IPP_CDBK_KMEANS_NUM	LBG algorithm with splitting of the most numerous clusters was used for the codebook building. The nearest codebook entry is located through a logarithmical search.						
<i>pCdbk</i>	Pointer to the codebook structure to be initialized.						

Description

The function `ippsCdbkInit` is declared in the `ippsr.h` file. This function initializes the structure that contains the codebook and additional information to be used for fast search. The structure is used during vector quantization by the `ippsSplitVQ` function. The Euclidean distance is used to measure the similarity between two vectors.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCdbk</i> or <i>pSrc</i> pointers are null.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>step</i> , <i>height</i> , or <i>cdbkSize</i> is less than or equal to 0; or <i>width</i> is greater than <i>step</i> ; or <i>cdbkSize</i> is greater than 8192.
<code>ippStsCdbkFlagErr</code>	Indicates an error when the <i>hint</i> value is incorrect or not supported.



NOTE. The only *hint* value that is currently supported for this function is `IPP_CDBK_FULL`. State memory address stored in *pCdbk* must be aligned to 32-bit word boundary.

CdbkInitAlloc

Initializes the codebook structure.

Syntax

```
IppStatus ippsCdbkInitAlloc_L2_16s(IppsCdbkState_16s** pCdbk, const Ipp16s*
    pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_L2_32f(IppsCdbkState_32f** pCdbk, const Ipp32f*
    pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_WgtL2_16s(IppsCdbkState_16s** pCdbk, const Ipp16s*
    pSrc, const Ipp16s* pWgt, int width, int step, int height, int cdbkSize,
    Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_WgtL2_32f(IppsCdbkState_32f** pCdbk, const Ipp32f*
    pSrc, const Ipp32f* pWgt, int width, int step, int height, int cdbkSize,
    Ipp_Cdbk_Hint hint);
```

Parameters

<i>pCdbk</i>	Pointer to the codebook structure to be created.
<i>pSrc</i>	Pointer to the source vector [<i>height</i> * <i>step</i>].
<i>pWgt</i>	Pointer to the weight vector [<i>width</i>].
<i>width</i>	Length of the input vectors.
<i>step</i>	Row step in the source vector <i>pSrc</i> .
<i>height</i>	Number of rows in the source vector <i>pSrc</i> .
<i>cdbkSize</i>	Size of the codebook.
<i>hint</i>	One of the following values:

IPP_CDBK_FULLL	The source data are entries of a codebook, <i>height</i> should be greater or equal to <i>cdbkSize</i> . The nearest codebook entry is located through a full search.
IPP_CDBK_KMEANS_LONG	LBG algorithm with splitting of the most extensional cluster is used for the codebook building. The nearest codebook entry is located through a logarithmical search.
IPP_CDBK_KMEANS_NUM	LBG algorithm with splitting of the most numerous clusters is used for the codebook building. The nearest codebook entry is located through a logarithmical search.

Description

The function `ippsCdbkInitAlloc` is declared in the `ippsr.h` file. This function initializes the structure that contains the codebook and additional information to be used for fast search. The structure is used during vector quantization by `ippsVQ` or `ippsSplitVQ` functions. The `ippsCdbkInitAlloc_L2` function uses the Euclidean distance to measure the similarity between two vectors, while the `ippsCdbkInitAlloc_WgtL2` function uses the weighted Euclidean distance for these purposes.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCdbk</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>step</i> , or <i>cdbkSize</i> is less than or equal to 0; or <i>cdbkSize</i> is greater than <i>height</i> ; or <i>width</i> is greater than <i>step</i> ; or <i>cdbkSize</i> is greater than 16383.
<code>ippStsCdbkFlagErr</code>	Indicates an error when the <i>hint</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.
<code>ippStsBadArgErr</code>	Indicates an error when one of <i>pWgt[i]</i> is less than 0.
<code>ippStsSmallerCodebook</code>	Indicates a warning when size of the created codebook is less than the <i>cdbkSize</i> argument value.

CdbkFree

Destroys the codebook structure.

Syntax

```
IppStatus ippCdbkFree_16s(IppsCdbkState_16s* pCdbk);  
IppStatus ippCdbkFree_32f(IppsCdbkState_32f* pCdbk);
```

Parameters

pCdbk Pointer to the codebook structure.

Description

The function `ippCdbkFree` is declared in the `ippsr.h` file. This function destroys the codebook structure and frees all memory associated with it.

Return Values

`ippStsNoErr` Indicates no error.
`ippStsNullPtrErr` Indicates an error when *pCdbk* pointer is NULL.

GetCdbkSize

Retrieves the number of codevectors in the codebook.

Syntax

```
IppStatus ippGetCdbkSize_16s(const IppsCdbkState_16s* pCdbk, int* pNum);  
IppStatus ippGetCdbkSize_32f(const IppsCdbkState_32f* pCdbk, int* pNum);
```

Parameters

pCdbk Pointer to the codebook structure.
pNum Pointer to the result number of codevectors.

Description

The function `ippsGetCdbkSize` is declared in the `ippsr.h` file. This function retrieves the number of codevectors in the codebook `pCdbk`. This number could be less than `cdbkSize` if the number of different vectors in `pSrc` is less than `cdbkSize`. The codebook structure `pCdbk` is initialized by the `ippsCdbkAlloc` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pCdbk</code> or <code>pNum</code> pointer is NULL.

GetCodebook

Retrieves the codevectors from the codebook.

Syntax

```
IppStatus ippsGetCodebook_16s(const IppsCdbkState_16s* pCdbk, Ipp16s* pDst, int
    step);
IppStatus ippsGetCodebook_32f(const IppsCdbkState_32f* pCdbk, Ipp32f* pDst, int
    step);
```

Parameters

<code>pCdbk</code>	Pointer to the codebook structure.
<code>pDst</code>	Pointer to the destination vector for codevectors [<code>pNum[0]*step</code>].
<code>step</code>	Row step in the destination vector <code>pDst</code> .

Description

The function `ippsGetCodebook` is declared in the `ippsr.h` file. This function retrieves the codevectors from the codebook structure `pCdbk` and stores them in the `pDst` vector with row step `step`.

The codebook structure `pCdbk` is initialized by the function `ippsCdbkAlloc`. The number of clusters can be obtained by the function `ippsGetCdbkSize`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCdbk</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or equal to 0 or <i>step</i> is less than <i>width</i> .

VQ

Quantizes the input vectors given a codebook.

Syntax

```
IppStatus ippvVQ_16s(const Ipp16s* pSrc, int step, Ipp32s* pIndx, int height,
    const IppsCdbkState_16s* pCdbk);
IppStatus ippvVQ_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx, int height,
    const IppsCdbkState_32f* pCdbk);
IppStatus ippvVQDist_16s32s_Sfs(const Ipp16s* pSrc, int step, Ipp32s* pIndx,
    Ipp32s* pDist, int height, const IppsCdbkState_16s* pCdbk,
    int scaleFactor);
IppStatus ippvVQDist_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx,
    Ipp32f* pDist, int height, const IppsCdbkState_32f* pCdbk);
```

Parameters

<i>pCdbk</i>	Pointer to the codebook structure.
<i>pSrc</i>	Pointer to the source vector [<i>height</i> * <i>step</i>].
<i>step</i>	Row step in the source vector <i>pSrc</i> .
<i>height</i>	Number of rows in the source vector <i>pSrc</i> .
<i>pIndx</i>	Pointer to the result index vector of the closest codevectors [<i>height</i>].
<i>pDist</i>	Pointer to the result quantization distances from the source vector [<i>height</i>].
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The functions `ippsVQ` and `ippsVQDist` are declared in the `ippsr.h` file. The function `ippsVQ` performs Vector Quantization (VQ) on the input vectors. The resulting indexes of the closest codevectors are stored in the vector `pIndx`. The function `ippsVQDist` also stores the distances (scaled with `scaleFactor` for the integer versions) to the output distance vector `pDist`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pCdbk</code> , <code>pSrc</code> , <code>pIndx</code> , or <code>pDist</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>step</code> or <code>height</code> is less than or equal to 0.

VQSingle_Sort, VQSingle_Thresh

Quantizes the input vector given a codebook and gets several closest clusters.

Syntax

```

IppStatus ippsVQSingle_Sort_32f(const Ipp32f* pSrc, Ipp32s* pIndx, const
    IppsCdbkState_32f* pCdbk, int num);

IppStatus ippsVQSingle_Sort_16s(const Ipp16s* pSrc, Ipp32s* pIndx, const
    IppsCdbkState_16s* pCdbk, int num);

IppStatus ippsVQDistSingle_Sort_32f(const Ipp32f* pSrc, Ipp32s* pIndx, Ipp32f*
    pDist, const IppsCdbkState_32f* pCdbk, int num);

IppStatus ippsVQDistSingle_Sort_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pIndx,
    Ipp32s* pDist, const IppsCdbkState_16s* pCdbk, int num, int scaleFactor);

IppStatus ippsVQSingle_Thresh_32f(const Ipp32f* pSrc, Ipp32s* pIndx, const
    IppsCdbkState_32f* pCdbk, Ipp32f val, int* pnum);

IppStatus ippsVQSingle_Thresh_16s(const Ipp16s* pSrc, Ipp32s* pIndx, const
    IppsCdbkState_16s* pCdbk, Ipp32f val, int* pnum);

IppStatus ippsVQDistSingle_Thresh_32f(const Ipp32f* pSrc, Ipp32s* pIndx,
    Ipp32f* pDist, const IppsCdbkState_32f* pCdbk, Ipp32f val, int* pnum);

```

```
IppStatus ippsVQDistSingle_Thresh_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pIndx,
      Ipp32s* pDist, const IppsCdbkState_16s* pCdbk, Ipp32f val, int* pnum, int
      scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pIndx</i>	Pointer to the destination indexes vector.
<i>pDist</i>	Pointer to the destination distances vector.
<i>pCdbk</i>	Pointer to the codebook structure.
<i>val</i>	Relative threshold value.
<i>num</i>	Number of closest clusters to search for each input vector.
<i>pnum</i>	Pointer to the number of clusters within threshold [1].
<i>scaleFactor</i>	Scaling factor for intermediate sums.

Discussion

The functions `ippsVQSingle_Sort` and `ippsVQSingle_Thresh` are declared in the `ippsr.h` file. These functions perform multiple vector quantization (VQ) for the input vector. Functions with `Sort` suffix provide *num* indexes of closest codebook centroids sorted in distance ascending order.

Functions with `Thresh` suffix provide indexes of codebook centroids with distance less than or equal to minimum multiplied by *val* and the number of such clusters. Functions with `Dist` suffix provide distance values of closest clusters. The length of vectors *pSrc* is equal to codevector length, the length of output vectors *pIndx* and *pDist* is equal to codebook size. Full search is done for all codebook types.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pIndx</i> , <i>pDist</i> , <i>pCdbk</i> or <i>pnum</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>num</i> is less than or equal to 0, or greater than the codebook size.
<code>ippStsBadArgErr</code>	Indicates an error when <i>val</i> is less than 1.

SplitVQ

*Quantizes a multiple-stream vector
given the codebooks.*

Syntax

```
IppStatus ippsSplitVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s* pDst, int
    dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_16s8u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst, int
    dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_16s1u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst, int
    dstBitStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_32f16s(const Ipp32f* pSrc, int srcStep, Ipp16s* pDst, int
    dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsSplitVQ_32f8u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst, int
    dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsSplitVQ_32f1u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst, int
    dstBitStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
```

Parameters

<i>pCdbks</i>	Pointer to the codebook structures [<i>nStream</i>].
<i>pSrc</i>	Pointer to the source vector [<i>height*srcStep</i>].
<i>srcStep</i>	Row step in the source vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the destination indexing vector [<i>height*dstStep</i>].
<i>dstStep</i>	Row step in the destination vector <i>pDst</i> .
<i>height</i>	Number of rows in the source and destination vectors.
<i>dstBitStep</i>	Row step in the destination vector (in bits).
<i>nStream</i>	Number of streams in the source vectors.

Description

The functions `ippsSplitVQ` is declared in the `ippsr.h` file. This function quantizes the multiple-stream vectors `pSrc` against given codebooks `pDbks`. The length of each stream is assumed to be equal to that of the corresponding codebook vectors. The outputs `pDst` are indexes to the codebook entries.

For functions with the `1u` suffix, the output indexes are packed in bits. Each stream takes the least number of bits sufficient to represent its codebook indexes.

See also [ippsFormVector](#), [ippsFormVectorVQ](#) functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pCdbk</code> , <code>pCdbk[k]</code> , <code>pSrc</code> , or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>srcStep</code> , <code>dstStep</code> , <code>height</code> , or <code>nStream</code> is less than or equal to 0; or the sum of the stream length is greater than <code>srcStep</code> ; or <code>nStream</code> is greater than <code>dstStep</code> for functions with the <code>16s</code> or <code>8u</code> suffix; or the number of bits sufficient to represent the indexes is greater than <code>dstStep</code> for functions with the <code>1u</code> suffix; or the codebook size is greater than 256 for functions with the <code>8u</code> suffix.

FormVectorVQ

Constructs multiple-stream vectors from codebooks, given indexes.

Syntax

```
IppStatus ippsFormVectorVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s*
    pDst, int dstStep, int height, const IppsCdbkState_16s** pDbks, int
    nStream);

IppStatus ippsFormVectorVQ_8u16s(const Ipp8u* pSrc, int srcStep, Ipp16s* pDst,
    int dstStep, int height, const IppsCdbkState_16s** pDbks, int nStream);
```



```
IppStatus ippsFormVectorVQ_1u16s(const Ipp8u* pSrc, int srcBitStep, Ipp16s*
    pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks, int
    nStream);

IppStatus ippsFormVectorVQ_16s32f(const Ipp16s* pSrc, int srcStep, Ipp32f*
    pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks, int
    nStream);

IppStatus ippsFormVectorVQ_8u32f(const Ipp8u* pSrc, int srcStep, Ipp32f* pDst,
    int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);

IppStatus ippsFormVectorVQ_1u32f(const Ipp8u* pSrc, int srcBitStep, Ipp32f* pDst,
    int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
```

Parameters

<i>pCdbks</i>	Pointer to the codebook structures [<i>nStream</i>].
<i>pSrc</i>	Pointer to the indexing vectors [<i>height*srcStep</i>].
<i>pDst</i>	Pointer to the constructed vectors [<i>height*dstStep</i>].
<i>srcStep</i>	Row step in the indexing vectors <i>pSrc</i> .
<i>srcBitStep</i>	Row step in the indexing vectors <i>pSrc</i> (in bits).
<i>dstStep</i>	Row step in the constructed vectors <i>pDst</i> .
<i>height</i>	Number of rows in the vectors <i>pSrc</i> .
<i>nStream</i>	Number of streams.

Description

The function `ippsFormVectorVQ` is declared in the `ippsr.h` file. This function constructs multiple-stream vectors *pDst* from the codebooks *pCdbks* given indexes *pSrc*. The length of each stream is assumed to be equal to that of the corresponding codebook vectors. For functions with the `1u` suffix, each stream index is assumed to be in a packed format. Each stream takes the number of bits sufficient to represent its codebook indexes. See also [ippsFormVector](#), [ippsSplitVQ](#) functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pCdbk</i> , <i>pCdbk[k]</i> , <i>pSrc</i> , or <i>pDst</i> pointer is NULL.

`ippStsSizeErr` Indicates an error when `srcStep`, `dstStep`, `height`, or `nStream` is less than or equal to 0;
or the codevector length sum is greater than `dstStep`;
or `nStream` is greater than `srcStep` for functions with the `16s` or `8u` suffix;
or the number of bits sufficient to represent the indexes is greater than `srcStep` for functions with the `1u` suffix.

Polyphase Resampling

The Intel IPP functions described in this section build, apply and free Kaiser-windowed polyphase filters for data resampling. Functions with `Fixed` suffix are intended for fixed rational resampling factor and provide faster speed. Functions without this suffix build universal resampling filter with linear interpolation of filter coefficients and allow to use a variable factor. The general description of the polyphase resampling algorithm can be found, for example, in *Multirate Digital Signal Processing* by R. Crochiere and L. Rabiner, Prentice Hall, 1983.

Some of these functions are used in the Intel IPP Speech recognition Sample. See *Speech Processing* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

ResamplePolyphaseInit

Initializes the structure for polyphase resampling without calculating the filter coefficients.

Syntax

```
IppStatus ippsResamplePolyphaseFixedInit_16s( IppsResamplingPolyphaseFixed_16s*  
    pSpec, int inRate, int outRate, int len, IppHintAlgorithm hint);  
IppStatus ippsResamplePolyphaseFixedInit_32f( IppsResamplingPolyphaseFixed_32f*  
    pSpec, int inRate, int outRate, int len, IppHintAlgorithm hint);
```

Parameters

<i>pSpec</i>	The pointer to the resampling state structure.
<i>inRate</i>	The input rate for fixed factor resampling.
<i>outRate</i>	The output rate for fixed factor resampling.
<i>len</i>	The filter length for fixed factor resampling.
<i>hint</i>	Suggests using specific code (must be equal to <code>ippAlgHintFast</code>). The values for the <i>hint</i> argument are described in Table 7-3, “Flag and Hint Arguments” .

Description

This function is declared in the `ippsr.h` file.

The function `ippsResamplePolyphaseFixedInit` initializes structures for data resampling with the factor equal to $inRate/outRate$ without calculating the filter coefficients. This function is designed for using pre-calculated filter coefficients for polyphase resampling.

First, you need to allocate memory for the structure. Then use the function `ippsResamplePolyphaseSetFixedFilter` to import filter coefficients. To calculate the memory size, filter length and the number of filters, use the `ippsResamplePolyphaseFixedGetSize` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>inRate</i> , <i>outRate</i> or <i>len</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates that <i>hint</i> is not equal to <code>ippAlgHintFast</code> .

ResamplePolyphaseGetSize

Gets the size of the polyphase resampling structure.

Syntax

```
IppStatus ippsResamplePolyphaseFixedGetSize_16s(int inRate, int outRate, int
    len, int* pSize, int* pLen, int* pHeight, IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseFixedGetSize_32f(int inRate, int outRate, int
    len, int* pSize, int* pLen, int* pHeight, IppHintAlgorithm hint);
```

Parameters

<i>inRate</i>	The input rate for fixed factor resampling.
<i>outRate</i>	The output rate for fixed factor resampling.
<i>len</i>	The filter length for fixed factor resampling.
<i>pSize</i>	The pointer to the variable that contains the size of the polyphase resampling structure.

<i>pLen</i>	The pointer to the variable that contains the real filter length.
<i>pHeight</i>	The pointer to the variable that contains the number of filters.
<i>hint</i>	Suggests using specific code (must be equal to <code>ippAlgHintFast</code>). The values for the <i>hint</i> argument are described in Table 7-3, “Flag and Hint Arguments” .

Description

The function `ippsResamplePolyphaseFixedGetSize` is declared in the `ippsr.h` file. This function determines the size required for the fixed rate polyphase resampling structure and associated storage, the filter length and the number of filters in the filter bank. The returned length of the filter is equal to $\min\{l \geq \text{len}, l \% 4\}$, the filter length for zero phase is greater by 1. These values can be used for export and import of fixed polyphase resampling filter.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> , <i>pLen</i> , or <i>pHeight</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>inRate</i> , <i>outRate</i> or <i>len</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates that <i>hint</i> is not equal to <code>ippAlgHintFast</code> .

ResamplePolyphaseSetFilter

Sets polyphase resampling filter coefficients.

Syntax

```
IppStatus ippsResamplePolyphaseSetFixedFilter_16s(
    IppsResamplingPolyphaseFixed_16s* pSpec, const Ipp16s* pSrc, int step, int
    height);

IppStatus ippsResamplePolyphaseSetFixedFilter_32f(
    IppsResamplingPolyphaseFixed_32f* pSpec, const Ipp32f* pSrc, int step, int
    height);
```

Parameters

<i>pSpec</i>	The pointer to the resampling state structure.
--------------	--

<i>pSrc</i>	The pointer to the input vector of filter coefficients.
<i>step</i>	The row step in <i>pSrc</i> vector.
<i>height</i>	The number of filters (the number of rows in <i>pSrc</i> vector).

Description

The function `ippsResamplePolyphaseSetFixedFilter` is declared in the `ippsr.h` file. This function imports pre-calculated filter coefficients into the polyphase resampling structure. If the *step* value is less than the filter length, trailing filter coefficients are zeroed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSpec</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates that <i>height</i> is greater than the number of filters in <i>pSpec</i> structure.

ResamplePolyphaseGetFilter

Gets polyphase resampling filter coefficients.

Syntax

```
IppStatus ippsResamplePolyphaseGetFixedFilter_16s(
    IppsResamplingPolyphaseFixed_16s* pSpec, const Ipp16s* pSrc, int step, int
    height);

IppStatus ippsResamplePolyphaseGetFixedFilter_32f(
    IppsResamplingPolyphaseFixed_32f* pSpec, const Ipp32f* pSrc, int step, int
    height);
```

Parameters

<i>pSpec</i>	The pointer to the resampling state structure.
<i>pSrc</i>	The pointer to the output vector of filter coefficients.
<i>step</i>	The row step in <i>pSrc</i> vector.

height The number of filters (the number of rows in *pSrc* vector).

Description

The function `ippsResamplePolyphaseSetFixedFilter` is declared in the `ippsr.h` file. This function exports filter coefficients from the polyphase resampling structure. If the *step* value is less than the filter length, only first *step* coefficients are exported.

Example 8-3 Export and import of the polyphase resampling filter bank

```
int inRate=16000; // input frequency
int outRate=8000; // output frequency
int history; // half of filter length
char fname[]="filter.flt\0"; // coefficient file name
{
int size,len,height;
FILE *file;
short *pFilter;
IppsresamplingPolyphaseFixed_16s *state;
history=(int)(64.0f*0.5*IPP_MAX(1.0,1.0/(double)outRate/(double)inRate))+1;
ippsResamplePolyphaseFixedInitAlloc_16s(&state,inRate,outRate,2*(history-1),
0.95f,9.0f,ippAlgHintFast);
ippsResamplePolyphaseFixedGetSize_16s(inRate,outRate,2*(history-1),&size,&len,
&height,ippAlgHintFast);
pFilter=ippsMalloc_16s(len*height);
ippsResamplePolyphaseGetFixedFilter_16s(state,pFilter,len,height);
file=fopen(fname,"wb");
fwrite(&size,sizeof(int),1,file);
fwrite(&len,sizeof(int),1,file);
fwrite(&height,sizeof(int),1,file);
fwrite(pFilter,sizeof(short),len*height,file);
fclose(file);
ippsFree(pFilter);
ippsResamplePolyphaseFixedFree_16s(state);
```

```

}
{
int size,len,height;
FILE *file;
short *pFilter;
IppsresamplingPolyphaseFixed_16s *state;
history=(int)(64.0f*0.5*IPP_MAX(1.0,1.0/(double)outRate/(double)inRate))+1;
file=fopen(fname,"rb");
fread(&size,sizeof(int),1,file);
fread(&len,sizeof(int),1,file);
fread(&height,sizeof(int),1,file);
pFilter=ippsMalloc_16s(len*height);
fread(pFilter,sizeof(short),len*height,file);
fclose(file);
state=(IppsresamplingPolyphaseFixed_16s*)ippsMalloc_8u(size);
ippsResamplePolyphaseFixedInit_16s(state,inRate,outRate,2*(history-1),
ippsAlgHintFast);
ippsResamplePolyphaseSetFixedFilter_16s((IppsresamplingPolyphaseFixed_16s*)
state,(const Ipp16s*) pFilter, len, height);
ippsFree(pFilter);
// use of polyphase filter
...
ippsFree(state);
}

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSpec</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates that <i>height</i> is greater than the number of filters in <i>pSpec</i> structure.

ResamplePolyphaseInitAlloc

Initializes the structure for polyphase data resampling.

Syntax

```
IppStatus ippsResamplePolyphaseInitAlloc_16s(IppsResamplingPolyphase_16s**  
    pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha,  
    IppHintAlgorithm hint);  
  
IppStatus ippsResamplePolyphaseInitAlloc_32f(IppsResamplingPolyphase_32f**  
    pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha,  
    IppHintAlgorithm hint);  
  
IppStatus ippsResamplePolyphaseFixedInitAlloc_16s(  
    IppsResamplingPolyphaseFixed_16s** pSpec, int inRate, int outRate, int len,  
    Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm hint);  
  
IppStatus ippsResamplePolyphaseFixedInitAlloc_32f(  
    IppsResamplingPolyphaseFixed_32f** pSpec, int inRate, int outRate, int len,  
    Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm hint);
```

Parameters

<i>pSpec</i>	The pointer to the resampling state structure to be created.
<i>window</i>	The size of the ideal lowpass filter window.
<i>nStep</i>	The discretization step for filter coefficients.
<i>rollf</i>	The roll-off frequency of the filter.
<i>alpha</i>	The parameter of the Kaiser window.
<i>inRate</i>	The input rate for resampling with fixed factor.
<i>outRate</i>	The output rate for resampling with fixed factor.
<i>len</i>	The filter length for resampling with fixed factor.
<i>hint</i>	Suggests using specific code. The values for the <i>hint</i> argument are described in Table 7-3, “Flag and Hint Arguments” .

Description

These functions are declared in the `ippsr.h` file.

The functions `ippsResamplePolyphaseInitAlloc` create structures for data resampling using the ideal lowpass filter

$$h[i] = \begin{cases} rollf, & i = 0 \\ \frac{\sin(rollf \cdot i\pi / nStep)}{i\pi / nStep}, & i \neq 0, |i| \leq window \cdot nStep, \end{cases}$$

$$0 < rollf \leq 1$$

and applies the Kaiser window with `alpha` parameter and width `window` to it.

This means that the values of the ideal lowpass filtering function are calculated (discretized) for all i such that $|i/nStep| \leq window$.

The structure created could be used to resample input samples by the function `ippsResample` with arbitrary resampling factor. In this case, filter coefficients for each output sample are calculated on the fly by using linear interpolation between two nearest values. The filter size depends on the resampling factor.

The functions `ippsResamplePolyphaseFixedInitAlloc` create structures for data resampling with the factor equal to `inRate/outRate`. If we denote the number of filters created in the `IppsResamplingPolyphaseFixed` structure for input and output frequencies by $fnum$, then

$$fnum = outRate / GCD(inRate, outRate)$$

where $GCD(a, b)$ stands for the greatest common divisor of a and b .

For example, if `inRate` = 8000 and `outRate` = 11025, then the number of filters will be $fnum = 11025 / GCD(8000, 11025) = 441$.

Values `window` and `nStep` in the formula above are chosen to provide the length $flen$ of each filter

$$flen = \min_{l \geq len, l \% 4 = 0} l$$

Note that for zero phase, the filter length is $flen+1$.

Functions with the “Fixed” suffix in their name pre-calculate filter coefficients for each phase and store them in the data structure for better performance. These functions can be used for cases when the ratio `inRate/outRate` is rational only. They can be considerably faster but may require large data structures for some input and output rates.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	<p>For <i>ippsResamplePolyphaseInitAlloc</i> functions: Indicates an error when <i>inRate</i>, <i>outRate</i>, <i>nStep</i> or <i>len</i> is less than or equal to 0.</p> <p>For <i>ippsResamplePolyphaseFixedInitAlloc</i> functions: Indicates an error when <i>inRate</i>, <i>outRate</i>, <i>nStep</i> or <i>len</i> is less than or equal to 0, or when $inRate/LCD * (outRate+1)/LCD$ is greater than 2^{29} (here LCD stands for the lowest common denominator of <i>inRate</i> and <i>outRate</i>).</p>
<code>ippStsBadArgErr</code>	Indicates an error when <i>rollf</i> is less than or equal to 0 or is greater than 1, or if <i>alpha</i> is less than 1, or if <i>window</i> is less than $2/nStep$.
<code>ippStsMemAllocErr</code>	Indicates a memory allocation error.

ResamplePolyphaseFree

Frees structure for polyphase data resampling.

Syntax

```

IppStatus ippsResamplePolyphaseFree_16s(IppsResamplingPolyphase_16s* pSpec);
IppStatus ippsResamplePolyphaseFree_32f(IppsResamplingPolyphase_32f* pSpec);
IppStatus ippsResamplePolyphaseFixedFree_16s(IppsResamplingPolyphaseFixed_16s*
    pSpec);
IppStatus ippsResamplePolyphaseFixedFree_32f(IppsResamplingPolyphaseFixed_32f*
    pSpec);

```

Parameters

<i>pSpec</i>	The pointer to the resampling state structure.
--------------	--

Description

The functions `ippsResamplePolyphaseFree` and `ippsResamplePolyphaseFixedFree` are declared in the `ippsr.h` file. These functions close the resampling structure by freeing all memory allocated for it by `ippsResamplePolyphaseInitAlloc` or `ippsResamplePolyphaseFixedInitAlloc` functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSpec</i> pointer is NULL.

ResamplePolyphase

Resamples input data using polyphase filters.

Syntax

```
IppStatus ippsResamplePolyphase_16s(const IppsResamplingPolyphase_16s* pSpec,
    const Ipp16s* pSrc, int len, Ipp16s* pDst, Ipp64f factor, Ipp32f norm,
    Ipp64f* pTime, int* pOutlen);

IppStatus ippsResamplePolyphase_32f(const IppsResamplingPolyphase_32f* pSpec,
    const Ipp32f* pSrc, int len, Ipp32f* pDst, Ipp64f factor, Ipp32f norm,
    Ipp64f* pTime, int* pOutlen);

IppStatus ippsResamplePolyphaseFixed_16s(const
    IppsResamplingPolyphaseFixed_16s* pSpec, const Ipp16s* pSrc, int len,
    Ipp16s* pDst, Ipp32f norm, Ipp64f* pTime, int* pOutlen);

IppStatus ippsResamplePolyphaseFixed_32f(const
    IppsResamplingPolyphaseFixed_32f* pSpec, const Ipp32f* pSrc, int len,
    Ipp32f* pDst, Ipp32f norm, Ipp64f* pTime, int* pOutlen);
```

Parameters

<i>pSpec</i>	The pointer to the resampling state structure.
<i>pSrc</i>	The pointer to the input vector.
<i>pDst</i>	The pointer to the output vector.
<i>len</i>	The number of input vector elements to resample.
<i>norm</i>	The norm factor for output samples.

<i>factor</i>	The resampling factor.
<i>pTime</i>	The pointer to the start time of resampling (in input vector elements). Keeps the input sample number and the phase for the first output sample from the next input data portion.
<i>pOutlen</i>	The number of calculated output vector elements.

Description

The functions `ippsResamplePolyphase` and `ippsResamplePolyphaseFixed` are declared in the `ippsr.h` file. These functions convert data from the input vector changing their frequency and compute all output samples that can be correctly calculated for the given input and the filter length. The ratio of output and input frequencies is defined by the *factor* argument for the function `ippsResamplePolyphase`. For the function `ippsResamplePolyphaseFixed`, this ratio is defined during creation of the resampling structure. The value `pTime[0]` defines the time value for which the first output sample is calculated.

The history data of filters are in the input vector with indexes less than `pTime[0]`. The history length is equal to $flen/2$ for function `ippsResamplePolyphaseFixed`, and

$$\left\lceil \frac{1}{2} window \cdot \max(1, 1/factor) \right\rceil + 1$$

for function `ippsResamplePolyphase`. Here *flen* is filter length and *window* is the size of the ideal lowpass filter window (see [ippsResamplePolyphaseInitAlloc](#)). The input vector should contain the same number of elements with indexes greater than `pTime[0]+len` for the right filter wing for the last element.

After function execution, the time value is updated and `pOutlen[0]` contains the number of calculated output samples.

The output samples are multiplied by $norm \cdot \min(1, factor)$ before saturation.

Example 8-4 Resampling of the input mono pcm file

```
void resampleIPP(
    int      inRate,    // input frequency
    int      outRate,   // output frequency
    FILE     *infd,     // input pcm file
    FILE     *outfd)    // output pcm file
{
```

```

short *inBuf,*outBuf;
int bufsize=4096;
int history=128;
double time=history;
int lastread=history;
int inCount=0,outCount=0,inLen,outLen;
IppsResamplingPolyphaseFixed_16s *state;

ippsResamplePolyphaseFixedInitAlloc_16s(&state,inRate,outRate,2*history,
                                         0.95f,9.0f,ippAlgHintAccurate);
inBuf=ippsMalloc_16s(bufsize+history+2);
outBuf=ippsMalloc_16s((int)((bufsize-history)*outRate/(float)inRate+2));

ippsZero_16s(inBuf,history);
while ((inLen=fread(inBuf+lastread,sizeof(short),bufsize-lastread,infd))>0)
{
    inCount+=inLen;
    lastread+=inLen;
    ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-history-(int)time,
                                    outBuf,0.98f,&time,&outLen);
    fwrite(outBuf,outLen,sizeof(short),outfd);
    outCount+=outLen;
    ippsMove_16s(inBuf+(int)time-history,inBuf,lastread+history-(int)time);
    lastread-=(int)time-history;
    time-=(int)time-history;
}
ippsZero_16s(inBuf+lastread,history);
ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-(int)time,
                                outBuf,0.98f,&time,&outLen);
fwrite(outBuf,outLen,sizeof(short),outfd);
outCount+=outLen;

printf("%d inputs resampled to %d outputs\n",inCount,outCount);
ippsFree(outBuf);
ippsFree(inBuf);
ippsResamplePolyphaseFixedFree_16s(state);
}

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSpec</i> , <i>pSrc</i> , <i>pDst</i> , <i>pTime</i> or <i>pOutLen</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates an error when <i>factor</i> is less than or equal to 0.

Advanced Aurora Functions

The Intel IPP functions discussed later in this section are implemented to support the speech processing and compression algorithm described in ETSI ES 202 050 V1.1.1 standard (see [ES202]).

The use of these functions is demonstrated in Intel IPP Speech Recognition Samples. See *Advanced Aurora Encoder-Decoder Sample* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

SmoothedPowerSpectrumAurora

Calculates smoothed magnitude of the FFT output.

Syntax

```

IppStatus ippsSmoothedPowerSpectrumAurora_16s(const Ipp16s* pSrc, Ipp16s* pDst,
int len);
IppStatus ippsSmoothedPowerSpectrumAurora_32f(const Ipp32f* pSrc, Ipp32f* pDst,
int len);
IppStatus ippsSmoothedPowerSpectrumAurora_32s64s_Sfs(Ipp32s* pSrc, Ipp64s*
pDst, Ipp32s len, int scaleFactor)
IppStatus ippsSmoothedPowerSpectrumAurora_32s_Sfs(Ipp32s* pSrc, Ipp32s* pDst,
Ipp32s len, int scaleFactor)

```

Parameters

<i>pSrc</i>	Pointer to the input vector in PERM format [<i>len</i>].
<i>pDst</i>	Pointer to the output vector [<i>len</i> /4+1].

<i>len</i>	The input vector length (multiple of 4).
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsSmoothedPowerSpectrumAurora` is declared in the `ippsr.h` file. This function calculates the smoothed square magnitude for the Fast Fourier Transform (FFT) output vector in PERM format ([ES202], 5.1.3-5.1.4):

$$pDst[0] = (pSrc[0]^2 + pSrc[2]^2 + pSrc[3]^2) / 2$$

$$pDst[i] = (pSrc[4 \cdot i]^2 + pSrc[4 \cdot i + 1]^2 + pSrc[4 \cdot i + 2]^2 + pSrc[4 \cdot i + 3]^2) / 2$$

$$pDst[len / 4] = pSrc[1]^2$$

where $0 < i < len / 4$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or <i>len</i> is not a multiple of 4.

NoiseSpectrumUpdate_Aurora

Updates the noise spectrum.

Syntax

```
IppStatus ippsNoiseSpectrumUpdate_Aurora_32f(const Ipp32f* pSrc, const Ipp32f*
    pSrcNoise, Ipp32f* pDst, int len);

IppStatus ippsNoiseSpectrumUpdate_Aurora_16s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pSrcNoise, Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsNoiseSpectrumUpdate_Aurora_32s_Sfs(const Ipp32s* pSrc, const
    Ipp32s* pSrcNoise, Ipp32s* pDst, int len, int scaleFactor);
```


Parameters

<i>pSrc</i>	Pointer to the input vector of the power spectral density mean [<i>len</i>].
<i>pSrcNoise</i>	Pointer to the input vector of the previous noiseless signal spectrum [<i>len</i>].
<i>pDst</i>	Pointer to the output vector of the improved transfer function [<i>len</i>].
<i>len</i>	The length of input and output vectors.
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsNoiseSpectrumUpdate` is declared in the `ippsr.h` file. This function updates the noise spectrum estimate on the second stage of noise reduction according to the formula ([ES202], 5.1.5):

$$pDst[i] = pSrcNoise[i] \cdot \left(0.9 + \frac{0.1 \cdot pSrc[i]}{pSrc[i] + pSrcNoise[i]} \cdot \left(1 + \frac{pSrcNoise[i]}{pSrcNoise[i] + 0.1 \cdot pSrc[i]} \right) \right)$$

for $0 \leq i < len$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pSrcNoise</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

WienerFilterDesign_Aurora

Calculates an improved transfer function of the adaptive Wiener filter.

Syntax

```
IppStatus ippsWienerFilterDesign_Aurora_32f(const Ipp32f* pSrc, const Ipp32f*
    pNoise, const Ipp32f* pDen, Ipp32f* pDst, int len);
IppStatus ippsWienerFilterDesign_Aurora_16s(const Ipp16s* pSrc, const Ipp16s*
    pNoise, const Ipp16s* pDen, Ipp16s* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the input vector of square roots of the power spectral density mean [<i>len</i>].
<i>pNoise</i>	Pointer to the input vector of square roots of the noise spectrum estimate [<i>len</i>].
<i>pDen</i>	Pointer to the input vector of square roots of the previous noiseless signal spectrum [<i>len</i>].
<i>pDst</i>	Pointer to the output vector of the improved transfer function [<i>len</i>] (in Q14).
<i>len</i>	The length of input and output vectors.

Description

The function `ippsWienerFilterDesign_Aurora` is declared in the `ippsr.h` file. This function processes the input vector according to the following formulas ([ES202], 5.1.5):

1. The noiseless signal is estimated:

$$p[i] = 0.98 \cdot pDen[i] + 0.02 \cdot \max(pSrc[i] - pNoise[i], 0)$$

2. The a priori Signal to Noise Ratio (SNR) is computed:

$$\eta[i] = p[i] / pNoise[i]$$

3. Then the filter transfer function

$$H[i] = \frac{\eta[i]}{1 + \eta[i]}$$

is used to improve the estimation of the noiseless signal spectrum:

$$p2[i] = H[i] \cdot pSrc[i]$$

4. The improved a priori SNR is calculated as:

$$\eta2[i] = \max(p2[i] / pNoise[i], 0.079432823)$$

5. The improved filter transfer function is obtained as:

$$pDst[i] = \frac{\gamma^2[i]}{1 + \eta^2[i]}$$

for $0 \leq i < len$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pNoise</i> , <i>pDen</i> , or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsBadArg</code>	Indicates an error when a negative or NaN argument of a square root operation is detected.

MelFBankInitAlloc_Aurora

Initializes the structure for performing the Mel-frequency filter bank analysis.

Syntax

```

IppStatus ippMelFBankInitAllocLow_Aurora_16s(IppsFBankState_16s** pFBank);
IppStatus ippMelFBankInitAllocLow_Aurora_32f(IppsFBankState_32f** pFBank);
IppStatus ippMelFBankInitAllocHigh_Aurora_16s(IppsFBankState_16s** pFBank);
IppStatus ippMelFBankInitAllocHigh_Aurora_32f(IppsFBankState_32f** pFBank);

```

Parameters

pFBank Pointer to the Mel-scale filter bank structure to be created.

Description

The function `ippMelFBankInitAlloc_Aurora` is declared in the `ippsr.h` file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis. The function with `Low` suffix builds a filter bank of 23 filters for 8 KHz data processing. The function with `High` suffix builds a filter bank of 3 filters for high frequency band of 16 KHz data processing ([[ES202](#)], 5.1.7).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pFBank</i> pointer is NULL.

TabCalculation_Aurora

Calculates filter coefficients for residual filter.

Syntax

```
IppStatus ippstabsCalculation_Aurora_16s(const Ipp16s* pSrc, Ipp16s* pDst);
IppStatus ippstabsCalculation_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst);
```

Parameters

<i>pSrc</i>	Pointer to the input vector of Mel-scaled filter bank output [10].
<i>pDst</i>	Pointer to the output filter coefficients vector [17].

Description

The function `ippstabsCalculation_Aurora` is declared in the `ippsr.h` file. This function calculates residual filter coefficients for the Mel-scaled filter bank output ([[ES202](#)], 5.1.9-5.1.10).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not equal to 23 or 26.

ResidualFilter_Aurora

Calculates a denoised waveform signal.

Syntax

```
IppStatus ippsResidualFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    Ipp16s* pTabs, int scaleFactor);

IppStatus ippsResidualFilter_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    Ipp32f* pTabs);
```

Parameters

<i>pTabs</i>	Pointer to the filter coefficients vector [17].
<i>pSrc</i>	Pointer to the input vector [96].
<i>pDst</i>	Pointer to the output vector [80].
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsResidualFilter_Aurora` is declared in the `ippsr.h` file.
 This function filters the input vector to get the low band and high band parts ([[ES202](#)], 5.1.10):

$$pDst[j] = \sum_{i=0}^{16} pTabs[i] \cdot pSrc[i + j] \quad , \quad 0 \leq j < 80$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pDst</i> , or <i>pTabs</i> pointer is NULL.

WaveProcessing_Aurora

Processes waveform data after noise reduction.

Syntax

```
IppStatus ippsWaveProcessing_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst);
IppStatus ippsWaveProcessing_Aurora_16s(const Ipp16s* pSrc, Ipp16s* pDst);
```

Parameters

pSrc Pointer to the input vector [200].
pDst Pointer to the output vector [200].

Description

The function `ippsWaveProcessing_Aurora` is declared in the `ippsr.h` file. This function processes the input vector according to the following formulas ([ES202], 5.2):

1. Smoothed Teager operator is calculated:

$$s[i] = \frac{1}{9} \sum_{k=-4}^4 s_T[i] \quad , \quad 0 \leq i < 200$$

where

$$s_T[i] = \left| pSrc[i]^2 - pSrc[\max(0, i-1)] pSrc[\min(199, i+1)] \right|$$

for $0 \leq i < 200$,

and $s[i] = 0$ for $i < 0$ and $200 \leq i$.

2. Peaks

$$p[l_{\min}], \dots, p[l_{\max}]$$

are found:

$$p[0] = \arg \max_{0 \leq i < 200} s[i]$$

$$p[l+1] = \arg \max_{\substack{p[l]+25 \leq k < p[l]+80 \\ k < 200}} s[k] , \quad l = 0, \dots, l_{\max} - 1$$

$$p[l-1] = \arg \max_{\substack{p[l]-25 \geq k > p[l]-80 \\ k \geq 0}} s[k] , \quad l = l_{\min} + 1, \dots, 0$$

3. Weighting function is calculated:

$$w[i] = \begin{cases} 1.2, & p[k] - 4 < i < p[k] + 0.8 \cdot]p[k+1] - p[k][\\ 1.0, & i = p[k] - 4, i = p[k] + 0.8 \cdot]p[k+1] - p[k][\\ 0.8 & \text{otherwise} \end{cases}$$

$$l_{\min} \leq k < l_{\max}$$

4. Output vector is computed as:

$$pDst[i] = w[i] \cdot pSrc[i] , \quad 0 \leq i < 200$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .

LowHighFilter_Aurora

Calculates low band and high band filters.

Syntax

```
IppStatus ippsLowHighFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDstLow,
    Ipp16s* pDstHigh, int len, const Ipp16s* pTabs, int tapsLen, int
    scaleFactor);

IppStatus ippsLowHighFilter_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDstLow,
    Ipp32f* pDstHigh, int len, const Ipp32f* pTabs, int tapsLen);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>len+tapsLen-1</i>].
<i>pDstLow</i>	Pointer to the low frequency band output vector [<i>len/2</i>].
<i>pDstHigh</i>	Pointer to the high frequency band output vector [<i>len/2</i>].
<i>len</i>	The input samples number (even).
<i>pTabs</i>	Pointer to the filter coefficients vector [<i>tapsLen</i>].
<i>tapsLen</i>	The filter taps number (even).
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsLowHighFilter_Aurora` is declared in the `ippsr.h` file. This function filters the input vector to get the low band and high band parts ([\[ES202\]](#), 5.5.1):

$$pDstLow[j] = \sum_{i=0}^{tapslen-1} pTabs[i] \cdot pSrc[i + 2 \cdot j]$$

$$pDstHigh[j] = \sum_{i=0}^{tapslen-1} pTabs[i] \cdot pSrc[i + 2 \cdot j] \cdot (-1)^{i+j} \quad , \quad 0 \leq j < len/2$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pDstLow</i> , <i>pDstHigh</i> , or <i>pTabs</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or <i>tapsLen</i> is less than or equal to 0, or <i>len</i> or <i>tapsLen</i> is not even.

HighBandCoding_Aurora

Codes and decodes the high frequency band energy values.

Syntax

```
IppStatus ippsHighBandCoding_Aurora_32f(const Ipp32f* pSrcHFB, const Ipp32f*
    pInSWP, const Ipp32f* pDSWP, Ipp32f* pDstHFB);
IppStatus ippsHighBandCoding_Aurora_32s_Sfs(const Ipp32s* pSrcHFB, const
    Ipp32s* pInSWP, const Ipp32s* pDSWP, Ipp32s* pDstHFB, int scaleFactor)
```

Parameters

<i>pSrcHFB</i>	Pointer to the input high frequency band energy vector [3].
<i>pInSWP</i>	Pointer to the input signal smoothed power spectrum vector [65].
<i>pDSWP</i>	Pointer to the denoised signal power spectrum vector [129].
<i>pDstHFB</i>	Pointer to the output coded high frequency band log energy vector [3].
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The function `ippsHighBandCoding_Aurora` is declared in the `ippsr.h` file. This function performs coding and decoding of high frequency band energies according to the following formulas ([[ES202](#)], 5.5.3):

1. Logarithm operation is applied to high frequency band energies:

$$S_{HFB}[k] = \ln pSrcHFB[k] , \quad 0 \leq k < 3$$

2. Auxiliary log energies of the input signal power spectrum are calculated:

$$S_{LFB_aux}[l] = \max \left(-50, \ln \left(\sum_{i=n1[l]}^{n1[l+1]-1} pInSWP[i] \right) \right) , \quad 0 \leq l < 3$$

$$n1[0] = 33, \quad n1[1] = 39, \quad n1[2] = 49, \quad n1[3] = 65.$$

3. Energies are coded as:

$$code[k, l] = S_{LFB_aux}[k] - S_{HFB}[l] , \quad 0 \leq k, l < 3$$

4. Auxiliary bands for decoding are calculated:

$$S_{Dec_aux}[l] = \max \left(-50, \ln \left(\frac{1}{2} \sum_{i=n2[l]}^{n2[l+1]-1} pDSWP[i] \right) \right) , \quad 0 \leq l < 3$$

$$n2[0] = 66, \quad n2[1] = 77, \quad n2[2] = 97, \quad n2[3] = 129.$$

5. Decoded high frequency band energies are computed as:

$$pDstHFB[k] = \sum_{l=0}^2 w[l] \cdot (S_{Dec_aux}[l] - code[l, k]) , \quad 0 \leq k < 3$$

$$w[0] = 0.1, \quad w[1] = 0.2, \quad w[2] = 0.7.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcHFB</i> , <i>pInSWP</i> , <i>pDSWP</i> , or <i>pDstHFB</i> pointer is <code>NUL</code> .
<code>ippStsBadArg</code>	Indicates an error when a non-positive or <code>NaN</code> argument of the logarithm operation is detected.

BlindEqualization_Aurora

Equalizes the cepstral coefficients.

Syntax

```
IppStatus ippsBlindEqualization_Aurora_32f(const Ipp32f* pRefs, Ipp32f* pCeps,
    Ipp32f* pBias, int len, Ipp32f val);

IppStatus ippsBlindEqualization_Aurora_16s(const Ipp16s* pRefsQ6, Ipp16s*
    pCeps, Ipp16s* pBias, int len, Ipp32s valQ6);
```

Parameters

<i>pRefs</i>	Pointer to the input vector of reference cepstrum [<i>len</i>].
<i>pCeps</i>	Pointer to the input and output vector of cepstrum [<i>len</i>].
<i>pBias</i>	Pointer to the input and output vector of bias [<i>len</i>].
<i>len</i>	The number of cepstral coefficients.
<i>val</i>	The log energy value.

Description

The function `ippsBlindEqualization_Aurora` is declared in the `ippsr.h` file. This function equalizes cepstral coefficients using the LMS algorithm ([[ES202](#)], 5.4):

$$step = 0.0087890625 \cdot \min(1, \max(0, val - 211 / 64))$$

$$pCeps[i] = pCeps[i] - pBias[i]$$

$$pBias[i] += step \cdot (pCeps[i] - pRefs[i]) \quad , \quad 0 \leq i < len$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pRefs</i> , <i>pCeps</i> , or <i>pBias</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

DeltaDelta_Aurora

Calculates the first and second derivatives according to ETSI ES 202 050 standard.

Syntax

```
IppStatus ippsDeltaDelta_Aurora_16s_D2Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    dstStep, int height, int deltaMode, int scaleFactor);

IppStatus ippsDeltaDeltaMul_Aurora_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
    pVal, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
    scaleFactor);

IppStatus ippsDeltaDelta_Aurora_32f_D2(const Ipp32f* pSrc, Ipp32f* pDst, int
    dstStep, int height, int deltaMode);

IppStatus ippsDeltaDeltaMul_Aurora_32f_D2(const Ipp32f* pSrc, const Ipp32f*
    pVal, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Parameters

<i>pSrc</i>	Pointer to the input feature sequence [<i>height</i> *14].
<i>pDst</i>	Pointer to the output feature sequence [<i>height</i> * <i>dstStep</i>].
<i>dstStep</i>	Length of the output feature in the output sequence <i>pDst</i> .
<i>height</i>	Number of feature vectors.
<i>deltaMode</i>	Execution mode.
<i>pVal</i>	Pointer to the delta coefficients vector [39].
<i>scaleFactor</i>	Refer to “Integer Scaling” in Chapter 2.

Description

The functions `ippsDeltaDelta_Aurora` and `ippsDeltaDeltaMul_Aurora` are declared in the `ippsr.h` file. These functions calculate full feature vectors according to ETSI ES 202 050 standard.

The input vectors of length 14 contain values $c_1, \dots, c_{12}, c_0, \ln E$. First, the input feature vectors are copied to the output sequence. Then the first and second derivatives are calculated ([[ES202](#)], 9.1-9.2).

The function implies the following constraints on the delta coefficients:

$val[j] = pVal[j]$ for `ippsDeltaDelta_Aurora` function
 $val[j] = 1$ for `ippsDeltaDeltaMul_Aurora` function.

$vel[-4] = -1.0, vel[-3] = -0.75, vel[-2] = -0.5, vel[-1] = -0.25, vel[0] = 0.0,$
 $vel[1] = 0.25, vel[2] = 0.5, vel[3] = 0.75, vel[4] = 1.0.$

$acc[-4] = 1.0, acc[-3] = 0.25, acc[-2] = -0.285714, acc[-1] = -0.607143,$
 $acc[0] = -0.714286, acc[1] = -0.607143, acc[2] = -0.285714, acc[3] = 0.25, acc[4] = 1.0.$

The execution mode `deltaMode` provides additional controls for the base feature copy and derivative calculation process. The admissible values of `deltaMode` and the corresponding function execution logic are the following:

1. `deltaMode` is equal to `IPP_DELTA_BEGIN|IPP_DELTA_END`

Functions perform the offline delta feature calculation. All base features are assumed available at the time of the calculation. The base features are copied from the input stream `pSrc` to the output stream `pDst` as follows:

$$pDst[i \cdot dstStep + j] = val[j] \cdot pSrc[i \cdot 14 + j] \quad , \quad 0 \leq j < 12.$$

$$pDst[i \cdot dstStep + 12] = val[12] \cdot (pSrc[i \cdot 14 + 12] \cdot 0.6/23 + pSrc[i \cdot 14 + 13] \cdot 0.4)$$

$$0 \leq i < height \quad . \quad (8.11)$$

Then the first and second derivatives are calculated as given by:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^4 vel[k] \cdot pDst[\max(0, \min(i + k, height - 1)) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^4 acc[k] \cdot pDst[\max(0, \min(i + k, height - 1)) \cdot dstStep + j]$$

for $0 \leq i < height, 0 \leq j < 13.$

2. `deltaMode` is equal to 0

Online delta feature calculation is performed. The input features are the current segment of a continuous stream. The function `ippsDeltaDelta_Aurora` calculates the partial delta features accordingly. First, the base features are copied as follows:

$$pDst[(i + 8) \cdot dstStep + j] = val[j] \cdot pSrc[i \cdot 14 + j] \quad , \quad 0 \leq j < 12.$$

$$pDst[(i+8) \cdot dstStep + 12] = val[12] \cdot (pSrc[i \cdot 14 + 12] \cdot 0.6/23 + pSrc[i \cdot 14 + 13] \cdot 0.4)$$

$$0 \leq i < height \quad . \quad (8.12)$$

Then the first and second derivatives are calculated as given by:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^4 vel[k] \cdot pDst[(i+k) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^4 acc[k] \cdot pDst[(i+k) \cdot dstStep + j]$$

for $0 \leq i - 4 < height$, $0 \leq j < 13$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the first and second derivatives in $pDst[k \cdot dstStep + 13 + l]$ are assumed available through a previous derivative calculation for $0 \leq j < 13$, $0 \leq i < 8$, $0 \leq k < 26$, $0 \leq l < 4$.

3. *deltaMode* is equal to `IPP_DELTA_BEGIN`

Functions perform the partial online delta feature calculation, where the beginning of the input stream is known. First, the base features are copied as in equation (8.11). Then, the first and second derivatives are calculated as follows:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^4 vel[k] \cdot pDst[\max(0, i+k) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^4 acc[k] \cdot pDst[\max(0, i+k) \cdot dstStep + j]$$

for $0 \leq i < height - 4$, $0 \leq j < 13$.

4. *deltaMode* is equal to `IPP_DELTA_END`

Functions perform the partial online delta feature calculation, where the ending of the input stream is known. First, the base features are copied as in equation (8.12). Then the first and second derivatives are calculated as given by:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^4 vel[k] \cdot pDst[\min(i + k, height - 1) \cdot dstStep + j]$$

and

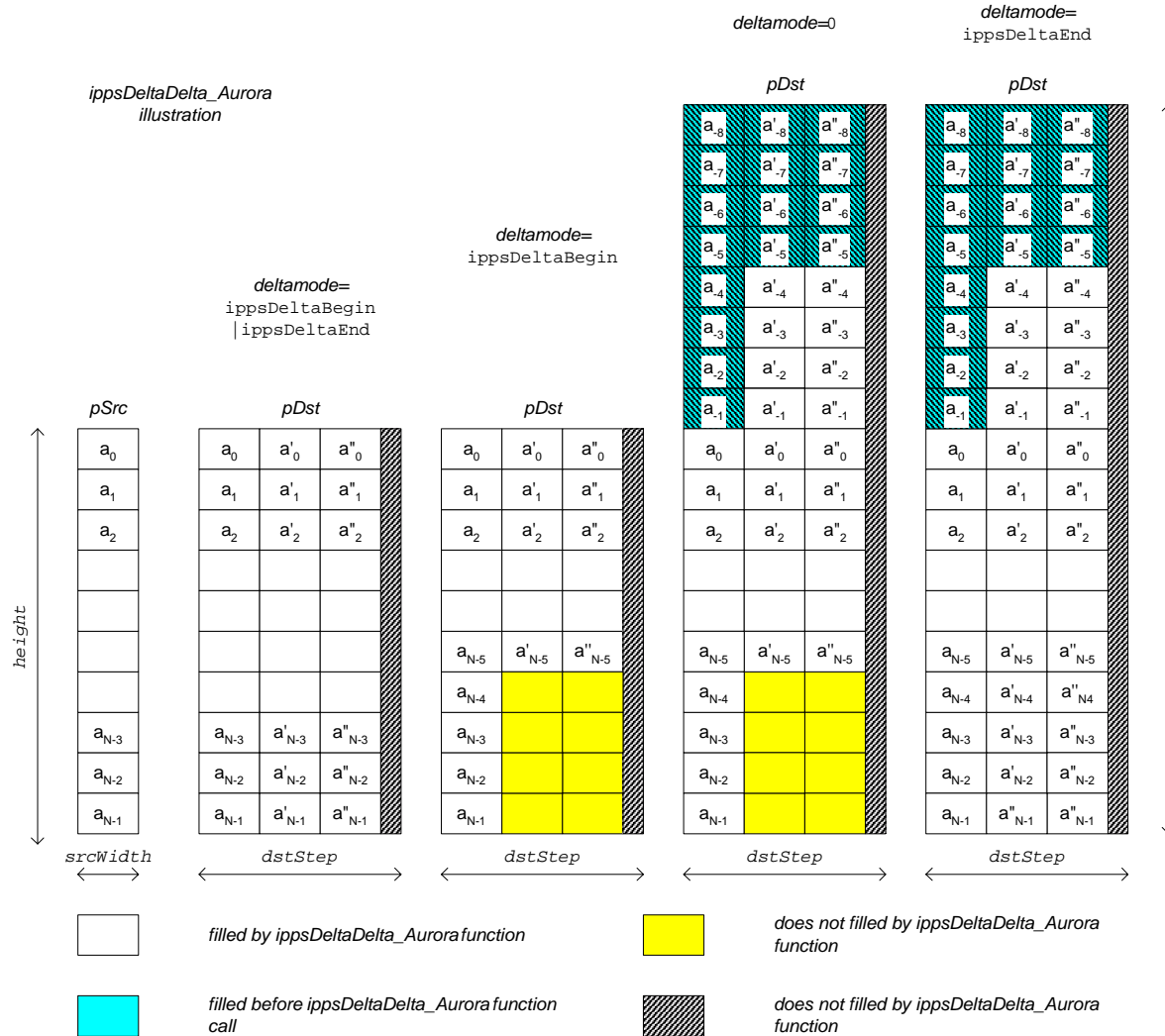
$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^4 acc[k] \cdot pDst[\min(i + k, height - 1) \cdot dstStep + j]$$

for $0 \leq i - 4 < height + 4$, $0 \leq j < 13$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the first and second derivatives in $pDst[k \cdot dstStep + 13 + l]$ are assumed available through a previous derivative calculation for $0 \leq j < 13$, $0 \leq i < 8$, $0 \leq k < 26$, $0 \leq l < 4$.

The following figure illustrates the above four delta calculation modes:

Figure 8-6 Execution Modes of ippsDeltaDelta_Aurora function



Return Values

ippStsNoErr

Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pDst</i> , or <i>pVal</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> is less than 0; or <i>height</i> is less than 8 when <i>deltaMode</i> is equal to IPP_DELTA_BEGIN; or <i>height</i> is equal to 0 when <i>deltaMode</i> is not equal to IPP_DELTA_END.
<code>ippStsStrideErr</code>	Indicates an error when <i>dstStep</i> is less than 39.

VADGetBufSize_Aurora

Queries the memory size for VAD decision.

Syntax

```
IppStatus ippvVADGetBufSize_Aurora_32f(int* pSize);
IppStatus ippvVADGetBufSize_Aurora_16s(int* pSize);
```

Parameters

<i>pSize</i>	Pointer to the output value of the memory size needed for VAD decision.
--------------	---

Description

The function `ippvVADGetBufSize_Aurora` is declared in the `ippsr.h` file. This function returns the size of memory that should be allocated by user. The memory block of `pSize[0]` bytes is used for VAD algorithm initialization by `ippvVadInit_Aurora` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is NULL.

VADInit_Aurora

Gets the VAD structure size.

Syntax

```
IppStatus ippSVADInit_Aurora_32f(char* pVADmem);
IppStatus ippSVADInit_Aurora_16s(char* pVADmem);
```

Parameters

pVADmem Pointer to the VAD decision memory.

Description

The function `ippSVADInit_Aurora` is declared in the `ippsr.h` file. This function initializes the VAD decision process.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when *pVADmem* pointer is NULL.

VADDecision_Aurora

Takes the VAD decision.

Syntax

```
IppStatus ippSVADDecision_Aurora_32f(const Ipp32f* pCoeff, const Ipp32f*
    pTrans, IppVADDecision_Aurora* pRes, int nbSpeechFrame, char* pVADmem);
IppStatus ippSVADDecision_Aurora_16s(const Ipp16s* pCoeff, const Ipp16s*
    pTrans, IppVADDecision_Aurora* pRes, int nbSpeechFrame, char* pVADmem);
```

Parameters

pCoeff Pointer to the input vector of Mel-warped Wiener filter coefficients [25].

<i>pTrans</i>	Pointer to the input vector of Wiener filter transfer function [64].
<i>pRes</i>	Pointer to VAD decision (“1” if voice is detected, “0” otherwise).
<i>nbSpeechFrame</i>	Speech frame hangover counter.
<i>pVADmem</i>	Pointer to the VAD decision memory.

Description

The function `ippsVADDecision_Aurora` is declared in the `ippsr.h` file. This function takes VAD decision for the input frame according to ([ES202], Annex A).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCoeff</i> , <i>pTrans</i> , <i>pRes</i> or <i>pVADmem</i> pointer is NULL.

VADFlush_Aurora

Takes VAD decision for zero input frame.

Syntax

```
IppStatus ippsVADFlush_Aurora_32f(IppVADDecision_Aurora* pRes, char* pVADmem);
IppStatus ippsVADFlush_Aurora_16s(IppVADDecision_Aurora* pRes, char* pVADmem);
```

Parameters

<i>pCoeff</i>	Pointer to the input vector of Mel-warped Wiener filter coefficients [25].
<i>pTrans</i>	Pointer to the input vector of Wiener filter transfer function [64].
<i>pRes</i>	Pointer to VAD decision (“1” if voice is detected, “0” otherwise).
<i>pVADmem</i>	Pointer to the VAD decision memory.

Description

The function `ippsVADDecision_Aurora` is declared in the `ippsr.h` file. This function takes VAD decision for the zero frame. It is used when speech is finished.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCoeff</i> , <i>pTrans</i> , <i>pRes</i> or <i>pVADmem</i> pointer is NULL.

Ephraim-Malah Noise Suppressor

This section describes the Intel IPP functions that can be combined to construct an implementation of the Ephraim-Malah Noise Suppressor (EMNS) originally described in [Eph84]. The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The EMNS is a frequency domain noise suppression algorithm that seeks to minimize the minimum mean squared error of the speech short-time spectral amplitude estimate.

Intel IPP EMNS primitives support the following features:

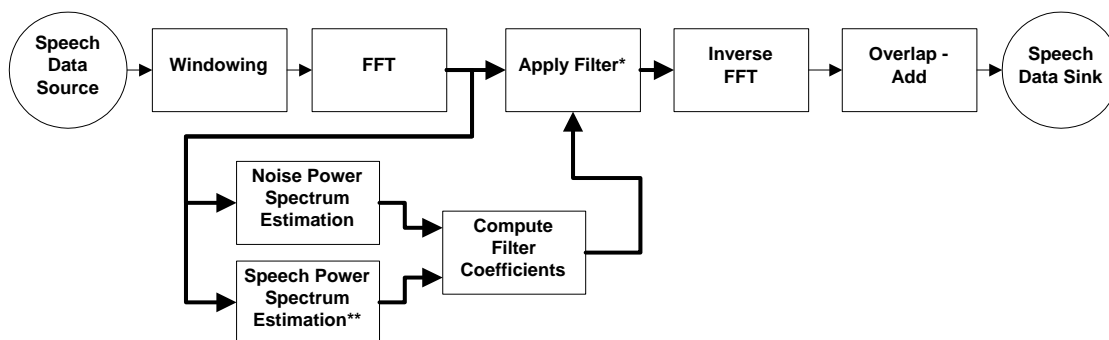
- Filter update
- Noise floor estimation

The use of these functions is demonstrated in Intel IPP Speech Recognition Samples. See *Audio Processing Sample* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Noise Suppressor Architecture

The major building blocks of the Ephraim-Malah Noise Suppressor are shown in [Figure 8-7](#), where the steps are given to apply noise reduction to a speech signal stream.

Figure 8-7 Major blocks in an Ephraim-Malah Noise Suppression System



* applying the filter consists of multiplying each FFT bin by a real-valued filter coefficient

** speech power spectral estimation is typically accomplished via spectral subtraction

Ephraim-Malah Noise Suppressor Details

Algorithm Steps

1. **New input block.** The most recently acquired $N/2$ input samples ζ_n and the previous $N/2$ input samples ζ_{n-1} make up the new input block \mathbf{z}_n .

$$\mathbf{z}_n = \begin{bmatrix} \zeta_{n-1} \\ \zeta_n \end{bmatrix}$$

2. **Windowed DFT.** The input block is multiplied by the square root of a window function. The window function is constrained such that when its first half is added to its second half, all values add to one. A convenient window function is the triangular window defined as

$$w(m) = \begin{cases} \frac{m+0.5}{N/2} & \text{for } m = 0, \dots, N/2 - 1 \\ 1 - w(m - N/2) & \text{for } m = N/2, \dots, N - 1 \end{cases}$$

The discrete Fourier transform of the input is calculated as follows.

$$\mathbf{Z}_n = \mathbf{F}(\mathbf{z}_n \bullet \sqrt{\mathbf{w}})$$

Here, \bullet denotes point-wise multiplication and $\sqrt{\mathbf{w}}$ denotes a vector containing the square root of the entries of \mathbf{w} . \mathbf{F} is the Fourier transform matrix with entries,

$$f(m,n) = \exp(-j2\pi mn/N), \text{ where } N \text{ is the size of the transform.}$$

3. **Update noisy speech PSD.** The noisy speech magnitude-squared spectral components are averaged to provide an estimate of the noisy speech power spectrum.

$$\mathbf{P}_n^z(k) = \beta_n \cdot |\mathbf{Z}_n(k)|^2 + (1 - \beta_n) \cdot \mathbf{P}_{n-1}^z(k)$$

The adaptive step size is defined as

$$\beta_n = \beta_{\min} + \rho_{n-1}^y (\beta_{\max} - \beta_{\min})$$

where $\beta_{\min} = 0.9$, $\beta_{\max} = 1.0$, and ρ_{n-1}^y is the likelihood of speech presence in bin k .

4. **Update clean speech PSD.** A coarse estimate of the clean speech power spectral components is obtained by spectral subtraction and averaging.

$$\mathbf{P}_n^y(k) = \alpha_n \cdot \left| \hat{\mathbf{Y}}_{n-1}(k) \right|^2 + (1 - \alpha_n) \cdot \psi_0(\mathbf{P}_n^z(k) - \mathbf{P}_{n-1}^v(k))$$

The thresholding operator ψ is defined as

$$\psi_c(x) = \begin{cases} c, & x \leq c \\ x, & x > c \end{cases}$$

The adaptive step size is defined as

$$\alpha_n = \alpha_{\min} + (1 - \rho_{n-1}^y)(\alpha_{\max} - \alpha_{\min})$$

where $\alpha_{\min} = 0.91$, $\alpha_{\max} = 0.95$, and ρ_{n-1}^y is the likelihood of speech presence in bin k .



NOTE. The previous frame's noise power spectral component is used in this calculation. If the noise floor estimator is independent of the rest of the algorithm, it may be possible to use the current frame's noise estimate instead. In this document, a dependency is assumed.

5. **Update Wiener filter weights.** It turns out that one of the quantities used to compute the Ephraim-Malah suppression rule is actually the Wiener filter (a different noise suppression rule). The Wiener filter weights are given by:

$$\mathbf{W}_n^y(k) = \psi_{W_{\min}} \left(\frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^y(k) + \mathbf{P}_{n-1}^v(k)} \right)$$

where W_{\min} corresponds to the threshold defined in [Cap94]. There it was recommended that a lower limit for a priori SNR

$$\mathfrak{R}_n^{prio}(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_{n-1}^v(k)}$$

of $\mathfrak{R}_{\min dB}^{prio} = -15.0$ dB be imposed to avoid musical noise. This corresponds to

$$W_{\min} = \frac{1}{1 + 10^{\frac{\mathfrak{R}_{\min dB}^{prio}}{10}}}$$

Note that if the Wiener filter is written in terms of the a priori SNR, then the Wiener filter calculation can be replaced by a table lookup. This may be advantageous on processors where division is expensive.

6. **Update a posteriori SNR.** The a posteriori signal to noise ratio for each frequency bin is defined as follows.

$$\mathfrak{R}_n^{post}(k) = \frac{\mathbf{P}_n^z(k)}{\mathbf{P}_{n-1}^v(k)}$$

7. **Update Ephraim-Malah filter weights.** The Ephraim-Malah filter weights are given by

$$\mathbf{H}_n^y(k) = \frac{1}{\mathfrak{R}_n^{post}(k)} \cdot M(\mathbf{W}_n^y(k) \mathfrak{R}_n^{post}(k))$$

where $M(\cdot)$ is the function

$$M(\theta) = \frac{1}{2} \cdot \sqrt{\pi\theta} \cdot e^{-\frac{\theta}{2}} \left[(1 + \theta) \cdot I_0\left(\frac{\theta}{2}\right) + \theta \cdot I_1\left(\frac{\theta}{2}\right) \right]$$

Note that this function can be replaced by a table lookup on processors where Bessel function evaluation is expensive

8. **Update noise PSD estimate.** A noise power spectral estimator must be employed to calculate $\mathbf{P}_n^v(k)$. One such estimator is the method of Martin

based on minimum statistics and optimal smoothing [Mar01].

9. **Update speech presence likelihood.** The probability of speech presence is not calculated directly. Rather, it is roughly approximated by the MMSE (Wiener) estimator of the overall speech energy.

$$\rho_n^y = \frac{\sum_{k=0}^{N/2} \mathbf{P}_n^y(k)}{\sum_{k=0}^{N/2} \mathbf{P}_n^y(k) + \sum_{k=0}^{N/2} \mathbf{P}_n^v(k)}$$

10. **Apply additional heuristic modifications to filter coefficients.**

The filter coefficients $\mathbf{H}_n^v(k)$ may be modified to improve perceptual speech quality or reduce perceptible musical tones. For example, to efficiently handle loud, low-pass noise such as that encountered in automotive environments, low-frequency filter coefficients (e.g., below 60 Hz) may be set to zero.

11. **Calculate filter output.** The filter output is defined as follows.

$$\hat{\mathbf{Y}}_n(k) = \mathbf{H}_n^y(k) \cdot \mathbf{Z}_n(k)$$

12. **Inverse DFT and overlap-add.** The time-domain filter output is defined as follows.

$$\hat{\mathbf{y}}_{n-1} = \begin{bmatrix} \mathbf{0}_{\frac{N}{2} \times \frac{N}{2}} & \mathbf{I}_{\frac{N}{2} \times \frac{N}{2}} \end{bmatrix} \cdot \sqrt{\mathbf{w}} \cdot \mathbf{F}^{-1} \hat{\mathbf{Y}}_{n-1} + \begin{bmatrix} \mathbf{I}_{\frac{N}{2} \times \frac{N}{2}} & \mathbf{0}_{\frac{N}{2} \times \frac{N}{2}} \end{bmatrix} \cdot \sqrt{\mathbf{w}} \cdot \mathbf{F}^{-1} \hat{\mathbf{Y}}_n$$

Note that the algorithm introduces a delay of N/2 samples in the output.

Data Structures

There is one structure associated with the MCRA noise floor estimator: `IppsMCRAParam`. This structure is used internally and should not be modified by the programmer.

Filter Update Primitives

FilterUpdateEMNS

Calculates the noise suppression filter coefficients.

Syntax

```
IppStatus ippsFilterUpdateEMNS_32s(const Ipp32s* pSrcWienerCoefsQ31, const Ipp32s*
    pSrcPostSNRQ15, Ipp32s* pDstFilterCoefsQ31, int len);
IppStatus ippsFilterUpdateEMNS_32f(const Ipp32f* pSrcWienerCoefs, const Ipp32f*
    pSrcPostSNR, Ipp32f* pDstFilterCoefs, int len);
```

Parameters

<i>pSrcWienerCoefsQ31</i>	Pointer to a real-valued vector containing the Q31 format Wiener filter coefficients ($0.0_{Q31} \leq pSrcWienerCoefsQ31[k] < 1.0_{Q31}$).
<i>pSrcWienerCoefs</i>	Pointer to a real-valued vector containing Wiener filter coefficients ($0.0 \leq pSrcWienerCoefs[k] < 1.0$).
<i>pSrcPostSNRQ15</i>	Pointer to a real-valued vector containing an estimate of the a posteriori signal to noise ratio in Q15 format ($0_{Q15} < pSrcPostSNRQ15 < 32768.0_{Q15}$).
<i>pSrcPostSNR</i>	Pointer to a real-valued vector containing an estimate of the a posteriori signal to noise ratio ($0 < pSrcPostSNR < 32768.0$).
<i>pDstFilterCoefsQ31</i>	Pointer to a real-valued vector containing the Q31 format filter coefficients ($0.0_{Q31} \leq pDstFilterCoefsQ31[k] < 1.0_{Q31}$).
<i>pDstFilterCoefs</i>	Pointer to a real-valued vector containing the filter coefficients ($0.0 \leq pDstFilterCoefs[k] < 1.0$).
<i>len</i>	Number of elements contained in input and output vectors ($0 < len < 65536$).

Description

The function `ippsFilterUpdateEMNS` is declared in the `ippsr.h` file. This function calculates the noise suppression filter coefficients. Three filter sizes are typically used: 65, 129, and 257 (corresponding to FFT sizes of 128, 256, and 512). These are recommended for sample rates $F_s \leq 11025$ Hz, $11025 \text{ Hz} < F_s \leq 20500$ Hz, and $20500 \text{ Hz} < F_s \leq 44100$ Hz, respectively. The noise suppression filter coefficients are the gains (scalar values between zero and one) that are applied to each FFT bin. These gains appear in the solution of the equation

$$\min E \left\{ \left(\left| \mathbf{Y}_n(k) \right| - \left| \hat{\mathbf{Y}}_n(k) \right| \right)^2 \right\}$$

where $\mathbf{Y}_n(k)$ is the k -th DFT component corresponding to the n -th block of speech samples and $\hat{\mathbf{Y}}_n(k)$ is an estimate of $\mathbf{Y}_n(k)$. Speech spectral components are assumed to have a Gaussian probability density. Since speech spectral components are not directly observable, the solution is written in terms of noisy observations, $\mathbf{Z}_n(k)$. The noise is assumed to be additive and Gaussian. The minimum mean squared error amplitude estimate is

$$\hat{\mathbf{Y}}_n(k) = \mathbf{H}_n(k) \cdot \mathbf{Z}_n(k)$$

where

$$\mathbf{H}_n(k) = \frac{\sqrt{\pi}}{2} \cdot \frac{\sqrt{\mathbf{W}_n^y(k) \cdot \mathfrak{R}_n^{post}(k)}}{\mathfrak{R}_n^{post}(k)} \cdot M(\mathbf{W}_n^y(k) \cdot \mathfrak{R}_n^{post}(k))$$

$$M(\theta) = e^{-\frac{\theta}{2}} \left[(1 + \theta) \cdot I_0\left(\frac{\theta}{2}\right) + \theta \cdot I_1\left(\frac{\theta}{2}\right) \right]$$

$$\mathbf{W}_n^y(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^y(k) + \mathbf{P}_{n-1}^v(k)}$$

$$\mathfrak{R}_n^{post}(k) = \frac{|\mathbf{Z}_n(k)|^2}{\mathbf{P}_{n-1}^v(k)}$$

I_0 and I_1 are Bessel functions, $\mathbf{P}_n^v(k)$ is an estimate of the k -th component of the power spectrum of the noise,

$\mathbf{P}_n^y(k)$ is an estimate of the k -th speech power spectral component.

The function `ippsFilterUpdate_EMNS` computes the filter coefficients $\mathbf{H}_n(k)$ according to the above equation. Power spectral estimates are typically computed using the `ippsAddWeighted` primitive.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcWienerCoefsQ31</code> , <code>pSrcPostSNRQ15</code> , <code>pDstFilterCoefsQ31</code> , <code>pSrcWienerCoefs</code> , <code>pSrcPostSNR</code> , or <code>pDstFilterCoefs</code> pointer is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error when <code>len</code> has an illegal value.

FilterUpdateWiener

Calculates the Wiener filter coefficients.

Syntax

```
IppStatus ippsFilterUpdateWiener_32s(const Ipp32s* pSrcPriorSNRQ15, Ipp32s*
    pDstFilterCoefsQ31, int len);

IppStatus ippsFilterUpdateWiener_32f(const Ipp32f* pSrcPriorSNR, Ipp32f*
    pDstFilterCoefs, int len);
```

Parameters

<i>pSrcPriorSNRQ15</i>	Pointer to a real-valued Q15 format vector containing an estimate of the a priori signal to noise ratio ($0.0_{Q15} < pSrcPriorSNRQ15[k] < 32768.0_{Q15}$).
<i>pSrcPriorSNR</i>	Pointer to a real-valued vector containing an estimate of the a priori signal to noise ratio ($0.0 < pSrcPriorSNRQ15[k] < 32768.0$).
<i>pDstFilterCoefsQ31</i>	Pointer to a real-valued vector containing the Q31 format filter coefficients ($0.0_{Q31} \leq pDstFilterCoefsQ31[k] < 1.0_{Q31}$).
<i>pDstFilterCoefs</i>	Pointer to a real-valued vector containing the filter coefficients ($0.0 \leq pDstFilterCoefsQ31[k] < 1.0$).
<i>len</i>	Number of elements contained in input and output vectors ($0 < len < 65536$).

Description

The function `ippsFilterUpdateWiener` is declared in the `ippsr.h` file. This function calculates the Wiener filter coefficients. Three filter sizes are typically used: 65, 129, and 257 (corresponding to FFT sizes of 128, 256, and 512). These are recommended for sample rates $F_s \leq 11025$ Hz, $11025 \text{ Hz} < F_s \leq 20500$ Hz, and $22050 \text{ Hz} < F_s \leq 44100$ Hz, respectively. The Wiener filter coefficients are the gains (scalar values between zero and one) that are applied to each FFT bin. These gains appear in the solution of the equation

$$\min E \left\{ \left| \mathbf{Y}_n(k) - \hat{\mathbf{Y}}_n(k) \right|^2 \right\}$$

where $\mathbf{Y}_n(k)$ is the k -th DFT component corresponding to the n -th block of speech

samples and $\hat{\mathbf{Y}}_n(k)$ is an estimate of $\mathbf{Y}_n(k)$. Speech spectral components are assumed to have a Gaussian probability density. Since speech spectral components are not directly observable, the solution is written in terms of noisy observations, $\mathbf{Z}_n(k)$.

The noise is assumed to be additive and Gaussian. The minimum mean squared error amplitude estimate is

$$\hat{\mathbf{Y}}_n(k) = \mathbf{W}_n^y(k) \cdot \mathbf{Z}_n(k)$$

where

$$\mathbf{W}_n^y(k) = \frac{1}{1 + \frac{1}{\Re_n^{prio}(k)}}$$

$$\Re_n^{prio}(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^v(k)}$$

This primitive implements a coarse approximation to $\mathbf{W}_n^y(k)$ in order to achieve very low execution time. This coarse approximation was found to be sufficient for most noise reduction applications. However, if high accuracy is desired one may choose to call `ippsDiv_32s_Sfs` with the proper parameters instead of `ippsFilterUpdateWiener_32s`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcPriorSNRQ15</code> , <code>pDstFilterCoefsQ31</code> , <code>pSrcPriorSNR</code> , or <code>pDstFilterCoefs</code> pointer is NULL.
<code>ippStsLengthErr</code>	Indicates an error when <code>len</code> has an illegal value.
<code>ippStsBadArgErr</code>	Indicates an error when at least one of the ‘a priori’ SNR array elements is negative. Zero filter coefficient is returned for such elements.

Noise Floor Estimation Primitives

GetSizeMCRA

Calculates the size in bytes required for the `IppMCRAState` state structure.

Syntax

```
IppStatus ippsGetSizeMCRA_32s(int nFFTSz, int* pDstSz);
IppStatus ippsGetSizeMCRA_32f(int nFFTSz, int* pDstSz);
```

Parameters

<i>nFFTSz</i>	Size of the FFT used for noise PSD estimations ($8 \leq nFFTSz \leq 8192$).
<i>pDstSz</i>	Pointer to the variable to contain the size in bytes.

Description

The function `ippsGetSizeMCRA` is declared in the `ippsr.h` file. This function calculates the size in bytes that should be allocated for the `IppMCRAState` state structure required by the `ippsUpdateNoisePSDMCRA` function. The function `ippsGetSizeMCRA` should be called before allocating memory and before calling `ippsInitMCRA`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDstSz</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>nFFTSz</i> has an Illegal value.

InitMCRA

Initializes the IppMCRAState state structure.

Syntax

```
IppStatus ippsInitMCRA_32s(int nSamplesPerSec, int nFFTSize, IppMCRAState*  
    pDst);  
IppStatus ippsInitMCRA_32f(int nSamplesPerSec, int nFFTSize, IppMCRAState32f*  
    pDst);
```

Parameters

<i>nSamplesPerSec</i>	Input sample rate ($8000 \leq nSamplesPerSec \leq 48000$).
<i>nFFTSize</i>	Size of the FFT used to for noise PSD estimation ($8 \leq nFFTSize \leq 8192$).
<i>pDst</i>	Pointer to the state structure.

Description

The function `ippsInitMCRA` is declared in the `ippsr.h` file. This function initializes the state structure for the `ippsUpdateNoisePSDMCRA` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates no error when <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>nFFTSize</i> has an illegal value.
<code>ippStsRangeErr</code>	Indicates an error when <i>nSamplesPerSec</i> is out of range.



NOTE. State memory address stored in *pDst* must be aligned to 32-bit word boundary.

AltInitMCRA

Allocates memory and initializes the IppMCRAState state structure.

Syntax

```
IppStatus ippsAltInitMCRA_32s(int nSamplesPerSec, int nFFTSIZE, IppMCRAState**
    ppDst);
IppStatus ippsAltInitMCRA_32f(int nSamplesPerSec, int nFFTSIZE, int
    nUpdateSamples, IppMCRAState32f* pDst);
```

Parameters

<i>nSamplesPerSec</i>	Input sample rate ($0 < nSamplesPerSec \leq 48000$).
<i>nFFTSIZE</i>	Size of the FFT used to for noise PSD estimation ($8 \leq nFFTSIZE \leq 8192$).
<i>nUpdateSamples</i>	Number of new samples per frame.
<i>ppDst</i>	Pointer to the pointer to the state structure.
<i>pDst</i>	Pointer to the state structure.

Description

The function `ippsInitAllocMCRA` is declared in the `ippsr.h` file. This function allocates memory and initializes the state structure for the `ippsUpdateNoisePSDMCRA` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>nFFTSIZE</i> has an illegal value.
<code>ippStsRangeErr</code>	Indicates an error when <i>nSamplesPerSec</i> is out of range.

UpdateNoisePSDMCRA

Re-estimates the noise power spectrum.

Syntax

```
IppStatus ippsUpdateNoisePSDMCRA_32s_I(const Ipp32s* pSrcNoisySpeech,  
    IppMCRAState* pSrcDstState, Ipp32s* pSrcDstNoisePSD);  
  
IppStatus ippsUpdateNoisePSDMCRA_32f_I(const Ipp32f* pSrcNoisySpeech,  
    IppMCRAState32f* pSrcDstState, Ipp32f* pSrcDstNoisePSD);
```

Parameters

<i>pSrcNoisySpeech</i>	Pointer to a real-valued vector containing the magnitude squared of the FFT of the noisy speech (<i>pSrcNoisySpeech</i> is greater than or equal to zero).
<i>pSrcDstState</i>	Pointer to the state structure.
<i>pSrcDstNoisePSD</i>	Pointer to the noise power spectrum vector.

Description

The function `ippsUpdateNoisePSDMCRA` is declared in the `ippsr.h` file. This function re-estimates the noise power spectrum given a new measurement of the magnitude squared noisy speech. The algorithm is based on the Minima Controlled Recursive Averaging approach described in [[Coh02](#)].

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcNoisySpeech</i> , <i>pSrcDstState</i> , or <i>pSrcDstNoisePSD</i> pointer is NULL.

Acoustic Echo Cancellor

This section describes the Intel IPP functions that can be combined to construct an Acoustic Echo Cancellor (AEC). The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The AEC consists of a frequency-domain adaptive filter algorithm and controller.

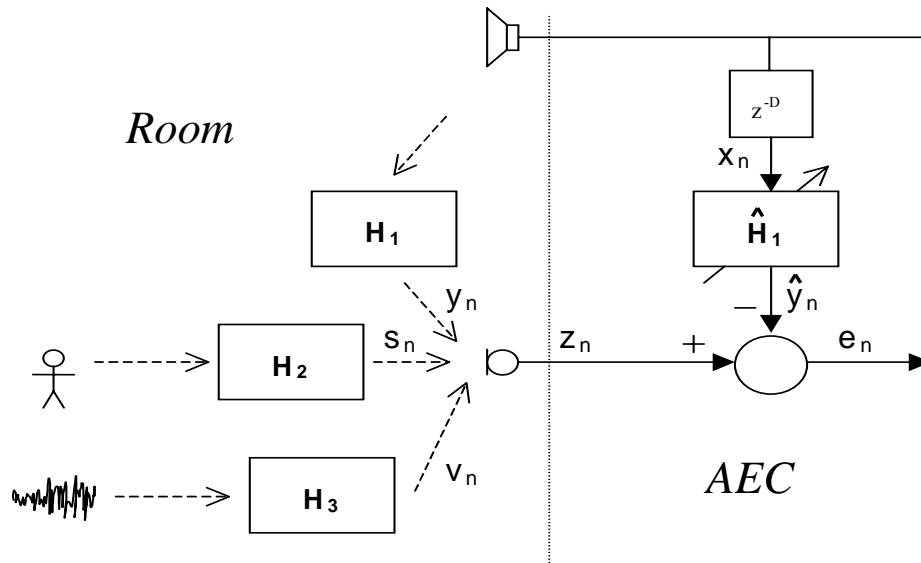
IPP AEC primitives support the following features:

- Filtering
- Filter coefficient
- Step size update
- AEC controller

Acoustic Echo Cancellor Architecture

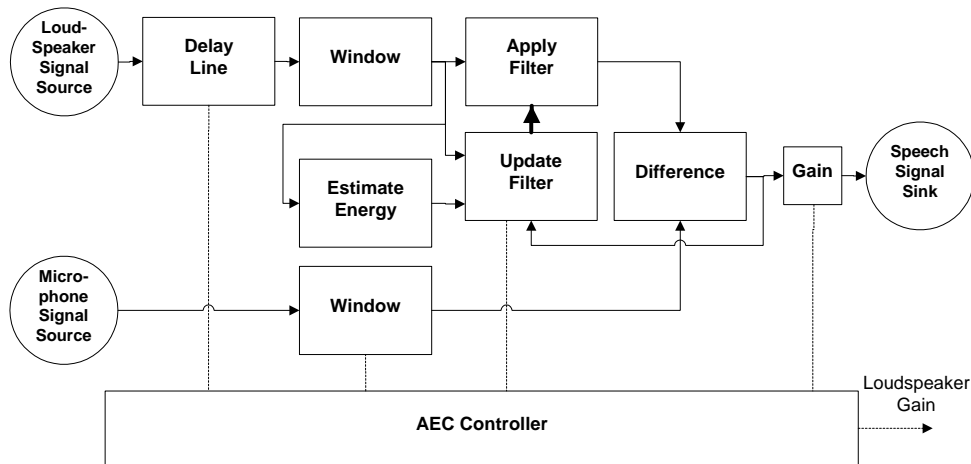
[Figure 8-8](#) shows a typical application of acoustic echo cancellation, the monophonic speakerphone. Here, audio is played into a room through a loudspeaker and captured at a microphone. The acoustic transfer function between the loudspeaker and the microphone is denoted \mathbf{H}_1 and the resulting signal at the microphone is denoted y_n where n is a time index. At the same time, a person is shown speaking into the microphone. The acoustic transfer function between the person's mouth and the microphone is denoted \mathbf{H}_2 and the resulting signal at the microphone is denoted s_n . A noise source (for example, a PC cooling fan) is also active in the room. The acoustic transfer function between the noise source and the microphone is denoted \mathbf{H}_3 and the resulting signal at the microphone is denoted v_n . The signals are assumed to combine additively at the microphone. The job of the AEC is to accurately estimate the transfer function \mathbf{H}_1 and delay D so that a filtered copy of the loudspeaker signal may be subtracted from the microphone signal, cancelling the echo.

Figure 8-8 A typical Acoustic Echo Cancellation Scenario



The major building blocks of the Acoustic Echo Canceller are shown in [Figure 8-9](#), where the steps are given to apply AEC to a speech signal stream.

Figure 8-9 Major blocks in an Acoustic Echo Cancellation System



Frequency Domain Block NLMS Adaptive Filter

Algorithm Steps

The frequency domain block normalized least mean square (NLMS) adaptive filter has a relatively simple computational structure and modest complexity [Ash94]. The algorithm consists of the following steps:

1. **New input block.** The most recently acquired $N/2$ input samples \mathbf{x}_n and the previous $N/2$ input samples \mathbf{x}_{n-1} make up the new input block. \mathbf{x}_n

$$\mathbf{x}_n = \begin{bmatrix} \mathbf{x}_{n-1} \\ \mathbf{x}_n \end{bmatrix}$$

2. **Input DFT.** The discrete Fourier transform of the input is calculated.

$$\mathbf{X}_n = \mathbf{F}\mathbf{x}_n$$

3. **Update input history.** The most recent L frequency-domain input blocks are retained.

$$\mathbf{X}_n = [\mathbf{X}_n \mathbf{X}_{n-1} \cdots \mathbf{X}_{n-L+1}]$$

4. **Update input energy estimate.** The energy at each bin $P_{xx}(k)$ may be estimated using a leaky average with $0 < \beta < 1$.

$$\hat{P}_{xx}(k) = (1 - \beta) \cdot \hat{P}_{xx}(k) + \beta \cdot |\mathbf{X}_n(k)|^2 \text{ for } k = 0, \dots, N/2$$

5. **Compute adaptive step sizes.** The adaptive step sizes are computed using the classical normalized LMS approach with $0 < \mu \ll 1$ and $P_{x\min} > 0$.

$$M(k) = \begin{cases} \frac{\mu}{\hat{P}_{xx}(k)}, & \hat{P}_{xx}(k) > P_{x\min} \\ \frac{\mu}{P_{x\min}}, & \hat{P}_{xx}(k) \leq P_{x\min} \end{cases} \text{ for } k = 0, \dots, N/2$$

6. **Compute the filter output.** The adaptive filter uses the low-latency structure described in [Ash94] where the filter impulse response is evenly divided into non-overlapping segments. In general, this corresponds to rewriting the convolution

$$y(m) = \sum_{i=0}^{I-1} h(i) \cdot x(m-i)$$

as

$$y(m) = \sum_{j=0}^{J-1} \sum_{k=0}^{I/J-1} h(k + j(I/J)) \cdot x(m - j(I/J) - k) = \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} h_j(k) \cdot x_j(m-k)$$

The filter output may be computed in the frequency domain as

$$\hat{\mathbf{Y}}_n(k) = \sum_{i=0}^{L-1} \mathbf{X}_{n-i}(k) \cdot \mathbf{H}_i(k) \text{ for } k = 0, \dots, N/2$$

where \mathbf{H}_i has been properly constrained (see step 9) such that the underlying convolution is linear.

7. **Constrain the error.** First the time-domain filter output is computed via an inverse discrete Fourier transform.

$$\hat{\mathbf{y}}_n = \mathbf{F}^{-1} \hat{\mathbf{Y}}_n$$

In the acoustic echo cancellation application, the error \mathbf{e}_n is the difference between the microphone input \mathbf{y}_n and the adaptive filter output.

$$\mathbf{e}_n = \mathbf{y}_n - \hat{\mathbf{y}}_n$$

The time-domain error must be constrained as follows

$$\mathbf{E}_n = \mathbf{F} \begin{bmatrix} \mathbf{0} \\ \boldsymbol{\varepsilon}_n \end{bmatrix}$$

to properly implement the adaptation step (see step 8) in the frequency-domain.

The n^{th} block of the echo canceller output is defined here as

$$\mathbf{\varepsilon}_n = [\mathbf{e}_n(N/2) \cdots \mathbf{e}_n(N-1)]^T$$

8. **Update adaptive filter.** The adaptive filter update is carried out in the frequency-domain. Segments of the adaptive filter response are updated independently by

$$\mathbf{H}_i = \mathbf{H}_i + \mathbf{M}_n \bullet \mathbf{X}_{n-i}^* \bullet \mathbf{E}_n \text{ for } i = 0, \dots, L-1$$

where \bullet denotes pointwise multiplication. Here $\mathbf{X}_{n-i}^* \bullet \mathbf{E}_n$ may be recognized as the i^{th} segment of the gradient.

9. **Constrain the adaptive filter response.** To exactly implement the linear convolution (filtering) operation in the frequency domain, it is necessary to apply a constraint. The filter response is constrained by

$$\mathbf{h}_i = \mathbf{F}^{-1} \mathbf{H}_i$$

$$\mathbf{h}_i = \mathbf{F}^{-1} \mathbf{H}_i$$

for $i = 0, \dots, L-1$. Here, the first half of each segment of the impulse response

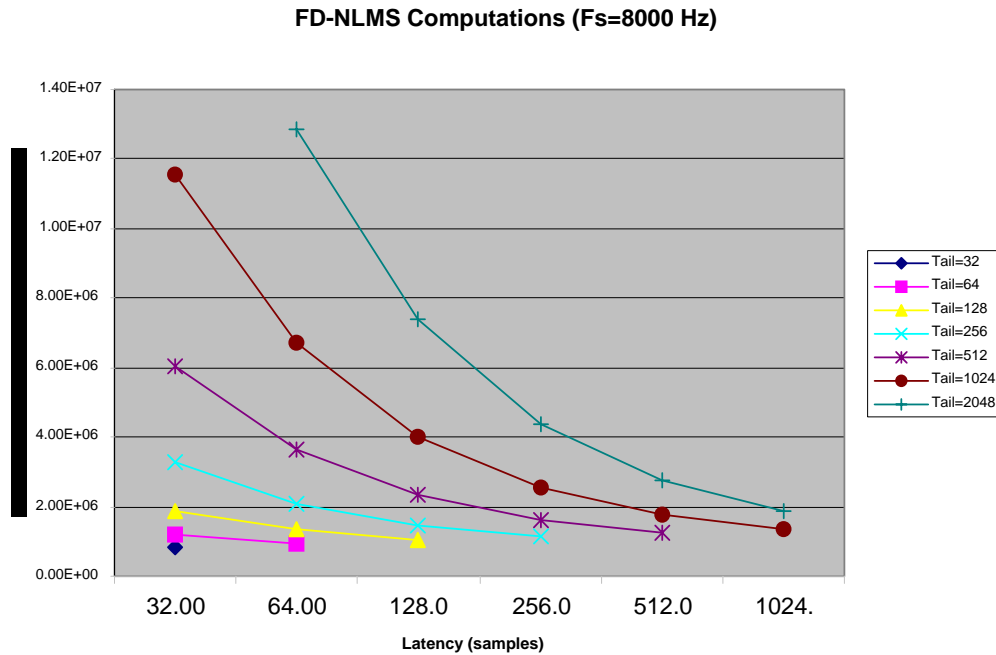
$$\mathbf{\eta}_i = [\mathbf{h}_i(0) \cdots \mathbf{h}_i(N/2-1)]^T$$

is retained and the rest set to zero.

Computational Complexity

The computational complexity of the adaptive filter is illustrated in [Figure 8-10](#). Here the trade-off between low latency and filter tail length are illustrated. For example, when latency is fixed at 8 milliseconds (64 samples per block at 8000 Hz sample rate), a tail length of 128 milliseconds requires around 7 MOPS (million operations per second). Reducing the latency to 4 milliseconds increases computations to 11.5 MOPS. Increasing tail length to 256 milliseconds increases computations to 13 MOPS.

Figure 8-10 Computational complexity of FD-NLMS adaptive filter



AEC Controller

To understand the need for an AEC controller, consider [Figure 8-8](#).

The adaptive filter \hat{H}_1 modifies its coefficients such that the filtered loudspeaker signal \hat{y}_n best matches the microphone signal z_n in the least squares sense.

When the loudspeaker signal is active, s_n and v_n are weak, and the loudspeaker-to-microphone transfer function H_1 is nearly linear, \hat{H}_1

will approach the true transfer function, H_1 . If the loudspeaker signal is active

and s_n or v_n are also active (a situation known as “double-talk”) then \hat{H}_1 will not converge to the proper solution. Even when s_n or v_n are weak, they may still cause the adaptive filter to diverge when the loudspeaker signal is also weak or silent. The AEC controller addresses these problems by adjusting the adaptive filter step size

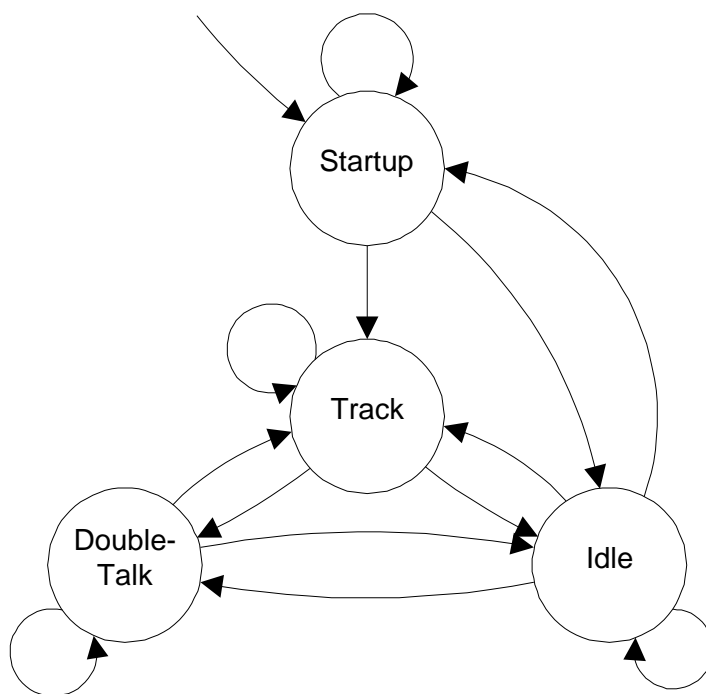
(μ in Step 5 above) such that the filter converges rapidly when the loudspeaker signal is present at

the microphone but does not diverge quickly during double-talk or insufficient excitation of the microphone by the loudspeaker. The controller also manages playback and output gain to block echo when the adaptive filter has diverged.

Algorithm Description

The AEC controller update primitive `ippsControllerUpdateAEC` is based on the simple state machine shown in [Figure 8-11](#)

Figure 8-11 State diagram of AEC controller.



State transitions are governed by simple full-band energy measurements:

$$E_n^e = \sum_{k=0}^{N/2-1} e_n^2(k)$$

error energy

$$E_n^z = \sum_{k=0}^{N/2-1} z_n^2(k)$$

microphone energy

$$E_n^x = \sum_{k=0}^{N/2-1} x_n^2(k)$$

loudspeaker energy

$$\bar{E}_n^z = (1 - \gamma) \cdot \bar{E}_{n-1}^z + \gamma \cdot E_n^z$$

smoothed microphone energy

$$\bar{E}_n^e = (1 - \gamma) \cdot \bar{E}_{n-1}^e + \gamma \cdot E_n^e$$

smoothed error energy

$$ERLE_n = \bar{E}_n^z / \bar{E}_n^e$$

echo return loss enhancement

The smoothing constant is $\gamma = 0.005$ when the block update rate is 8 milliseconds (e.g., 64 samples at 8000 Hz sample rate) and is adjusted to achieve the same smoothing rate at other block update rates. Four conditions are defined using the energy measurements:

Converged (C) $ERLE_n > T^{ERLE}$

Receive Active (R) $E_n^x > T_n^x$

Mic Active (M) $E_n^z > T_n^z$

No Double-talk (N) $E_n^x \cdot G_{xz} < E_n^z$

The thresholds are set as follows: the ERLE threshold T^{ERLE} is 2.0 (3 dB),

the loudspeaker activity threshold T_n^x is $6.0 \cdot \min\{E_n^x, E_{n-1}^x, \dots, E_{n-\text{window size}}^x\}$

(8 dB above the local minimum), the microphone activity threshold T_n^z is

$$6.0 \cdot \min\{E_n^z, E_{n-1}^z, \dots, E_{n-\text{window size}}^z\} \quad (8 \text{ dB above the local minimum}),$$

and the external gain between the loudspeaker and microphone G_{xz} is 0.25 (-6 dB).

The minimum-tracking window size corresponds to 2 seconds.

State transitions are triggered by the four conditions according to the table below.

The step size, loudspeaker gain, and output gain are adjusted as shown.

C	R	M	N	Current State	Next State	μ	α	β
F	T	T	T	Startup	Startup	$4\mu_c$	0.0	1.0
F	T	F	X	Startup	Startup	μ_c	0.0	1.0
F	T	T	F	Startup	Startup	0.0	1.0	0.0
X	F	T	X	Startup	Startup	0.0	1.0	0.0
X	F	F	X	Startup	Startup	0.0	0.5	0.5
T	T	T	T	Startup	Track	$4\mu_c$	1.0	1.0
T	T	T	F	Startup	Track	μ_c	1.0	1.0
T	T	F	T	Startup	Track	μ_c	0.0	1.0
T	T	F	F	Startup	Idle	0.0	0.5	0.5
T	T	T	X	Track	Track	$2\mu_c$	1.0	1.0
X	T	F	X	Track	Track	μ_c	1.0	1.0
F	T	T	X	Track	Double-talk	0.0	1.0	1.0
X	F	X	X	Track	Idle	0.0	0.5	0.5
X	F	T	X	Idle	Idle	0.0	1.0	1.0
X	F	F	X	Idle	Idle	0.0	0.5	0.5

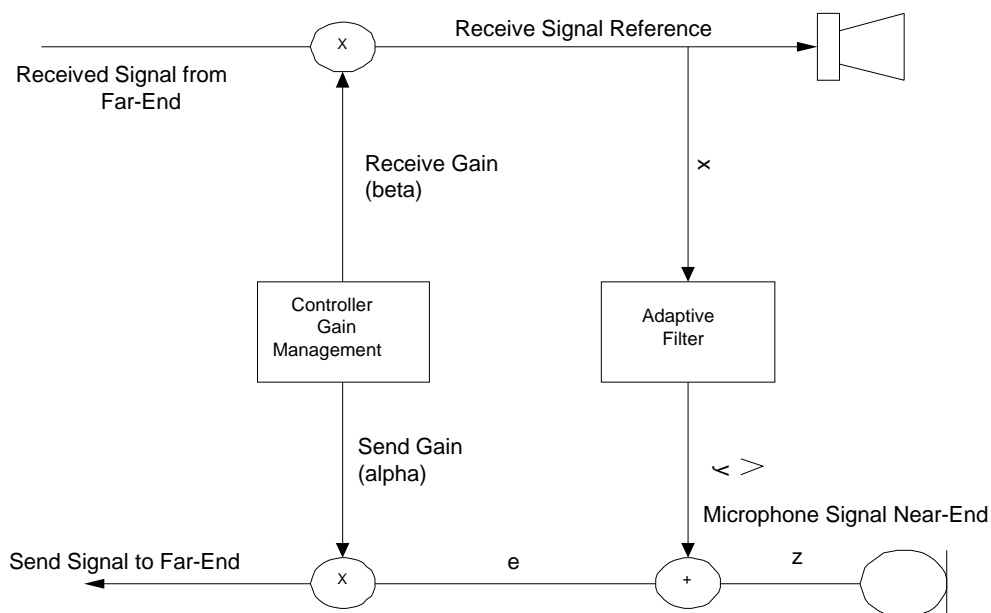
T	T	T	T	Idle	Track	$2\mu_c$	1.0	1.0
T	T	T	F	Idle	Track	$2\mu_c$	1.0	1.0
X	T	F	X	Idle	Track	$2\mu_c$	1.0	1.0
X	T	T	F	Idle	Double-talk*	0.0	1.0	1.0
F	T	T	X	Double-talk	Double-talk	0.0	1.0	1.0
T	T	T	T	Double-talk	Track	$2\mu_c$	1.0	1.0
T	T	T	F	Double-talk	Track	μ_c	1.0	1.0
X	T	F	X	Double-talk	Track	μ_c	1.0	1.0
T	T	T	F	Double-talk	Track*	0.0	1.0	1.0
X	F	T	X	Double-talk	Idle	0.0	1.0	1.0
X	F	F	X	Double-talk	Idle	0.0	0.5	0.5

Transitions marked with an “*” are redundant but take precedence when the ERLE exceeds the threshold for the “converged” (C) condition but is still below 40% of the peak ERLE in dB, that is,

$$\log(T^{ERLE}) < \log(ERLE_n) < 0.4 \cdot \log(\max\{ERLE_n, ERLE_{n-1}, \dots, ERLE_{n-window\ size}\})$$

The values α and β in the table correspond to the send and receive gain targets, respectively. The instantaneous gains are modified gradually to avoid audible flutter. When one of the gain targets changes, the instantaneous gain is increased or decreased linearly to meet the target in exactly 100 milliseconds. [Figure 8-12](#) illustrates that application of the send and receive gain in an AEC system.

Figure 8-12 Application of AEC send and receive gains by the AEC controller.



Data Structures

The AEC Controller requires access to many parameters. These parameters are stored in the following structures:

```
typedef struct {
    Ipp16s *pMicrophone;           /* pointer to mic samples */
    Ipp16s *pLoudspeaker;          /* pointer to speaker samples */
    Ipp16s *pError;                /* pointer to error samples */
    Ipp32s *pAFInputPSD;           /* pointer to filter input PSD */
    Ipp32sc **ppAFCoefs;           /* pointer to filter segment array */
    Ipp32s muQ31;                  /* fixed step size (Q31 value in [0,1)) */
    Ipp32s AECOutGainQ30;          /* AEC output gain (Q30 value in [0,1]) */
    Ipp32s speakerGainQ30;         /* loudspeaker gain (Q30 value in [0,1]) */
} /*
```

```

int numSegments;           /* number of segments of filter tail */
int numFFTBins;            /* number of FFT bins (FFTSize / 2 + 1) */
int numSamples;            /* mic, error, loudspeaker frame size */
int sampleRate;            /* sample rate (Hertz) */

} IppAECNLMSPParam;

typedef struct {
    Ipp32f *pMicrophone;    /* pointer to mic samples */
    Ipp32f *pLoudspeaker;   /* pointer to speaker samples */
    Ipp32f *pError;         /* pointer to error samples */
    Ipp32f *pAFInputPSD32f; /* pointer to filter input PSD32f */
    Ipp32fc **ppAFCoefs32f; /* pointer to filter segment array */
    Ipp32f mu;              /* fixed step size (value in (0,1)) */
    Ipp32f AECOutGain;      /* AEC output gain (Q30 value in [0,1]) */
    Ipp32f speakerGain;     /* loudspeaker gain (Q30 value in [0,1]) */
    int numSegments;        /* number of segments of filter tail */
    int numFFTBins;         /* number of FFT bins (FFTSize / 2 + 1) */
    int numSamples;         /* mic, error, and loudspeaker frame size */
    int sampleRate;         /* sample rate */
} IppAECNLMSPParam32f;

```

Note that `numSamples` represents the size of the non-overlapping part of the frames. The valid ranges for the elements of the `IppAECNLMSPParam` structure are shown in the table below.

Member	Range
<code>pMicrophone</code>	Samples are in $[-2^{15}, +2^{15}]$.
<code>pLoudspeaker</code>	Samples are in $[-2^{15}, +2^{15}]$.
<code>pError</code>	Samples are in $[-2^{15}, +2^{15}]$.
<code>pAFInputPSD</code>	PSD coefficients are in range $[0, 2^{31}]$.
<code>ppAFCoefs</code>	Real and imaginary parts of coefficients are in $[-2^{31}, +2^{31}]$.
<code>muQ31</code>	Q31 scalar in range $[0, 1]$.
<code>AECOutGainQ30</code>	Q30 scalar in range $[0, 1]$.

speakerGainQ30	Q30 scalar in range [0 , 1].
numSegments	In range [1 , 255].
numFFTBins	Is one of {17, 33, 65, 129, 257, 513, 1025,2049, 4097}.
numSamples	In range [32 , 4096]
sampleRate	In range [8000 , 48000]
pAFInputPSD32f	Real valued PSD coefficients
ppAFCoefs32f	Real and imaginary parts of coefficients
mu	Real value in range [0 , 1).
AECOutGain	Real value in range [0 , 1].
speakerGain	Real value in range [0 , 1].

Filter tail lengths of up to 2 seconds are supported. The pointers in the structure should always point to the most recent block or coefficient set. The controller keeps its internal state in an additional structure called `IppAECtrlState`.

The `IppAECtrlState` structure is used internally and should not be modified by the programmer.

The structure `IppAECScaled32s` provides a representation for scaled 32-bit signed integers.

```
typedef struct {
    Ipp32s val;
    Ipp32s sf;
} IppAECScaled32s;
```

A variable `x` of type `IppAECScaled32s` represents the value $x.val * 2^{x.sf}$. The structure `IppAECScaled32s` provides for efficient computations on the Intel® PCA processor family compared to floating point, at the cost of increased storage. Typically, the `val` field is “left justified”, meaning that it has been left shifted such that

$$2^{30} \leq x.val < 2^{31} \quad \text{or} \quad -2^{30} < x.val \leq -2^{31}.$$

Filter Primitives

FilterAECNLMS

Computes the frequency-domain adaptive filter output.

Syntax

```
IppStatus ippsFilterAECNLMS_32sc_Sfs(Ipp32sc** ppSrcSignalIn,
    Ipp32sc** ppSrcCoefs, Ipp32sc* pDstSignalOut, int numSegments,
    int len, int scaleFactor);
```

Parameters

<i>ppSrcSignalIn</i>	Pointer to an array of pointers to the most recent input blocks(e.g., X_n , X_{n-1} , ..., X_{n-L+1}). These are the complex-valued vectors that contain the FFT of the input signal. The argument <i>ppSrcSignalIn</i> is a two-dimensional complex vector of size $[numSegments][len]$.
<i>ppSrcCoefs</i>	Pointer to an array of pointers to the filter coefficients vectors. These are the complex-valued vectors containing the filter coefficients. The argument <i>ppSrcCoefs</i> is a two-dimensional complex vector of size $[numSegments][len]$.
<i>pDstSignalOut</i>	Pointer to the complex-valued filter output vector.
<i>numSegments</i>	The number of filter segments (L) ($0 < numSegments < 256$).
<i>len</i>	Number of elements contained in the input and output vectors and each filter segment ($0 < len \leq 4097$).
<i>scaleFactor</i>	Saturation fixed scale factor ($-32 < scaleFactor < 32$)

Description

The function `ippsFilterAECNLMS` is declared in the `ippalign.h` file. This function implements [step 6](#) of the above algorithm to compute the frequency-domain adaptive filter output.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when <i>ppSrcSignalIn</i> , <i>ppSrcCoefs</i> , or <i>pDstSignalOut</i> pointer is NULL.
<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> has an illegal value.
<code>ippStsRangeErr</code>	Indicates an error when <i>numSegments</i> or <i>scaleFactor</i> is out of range.

Filter Update Primitives

CoefUpdateAECNLMS

Updates the adaptive filter coefficients.

Syntax

```
IppStatus ippsCoefUpdateAECNLMS_32sc_I(const IppAECScaled32s* pSrcStepSize,
    const Ipp32sc** ppSrcFilterInput, const Ipp32sc* pSrcError,
    Ipp32sc** ppSrcDstCoefs, int numSegments, int len, int scaleFactorCoef);
```

Parameters

<i>pSrcStepSize</i>	For integer function, this is the pointer to the scaled integer step size vector. It is recommended that the <i>val</i> field is left justified, that is, $230 \leq pSrcStepSize[k].val < 231$ and the <i>sf</i> field is restricted to the range $(-64 \leq pSrcStepSize[k].sf < 64)$. The dimension of <i>pSrcStepSize</i> is <i>[len]</i> . For floating point function, this is the pointer to double step size vector.
<i>ppSrcFilterInput</i>	Pointer to an array of pointers to the most recent input blocks (e.g., $X_n, X_{n-1}, \dots, X_{n-L+1}$). These are the complex-valued vectors that contain the FFT of the input signal. The dimension of <i>ppSrcFilterInput</i> is <i>[numSegments][len]</i> .
<i>pSrcError</i>	Pointer to the complex-valued vector containing the filter error. The dimension of <i>pSrcError</i> is <i>[len]</i> .
<i>ppSrcDstCoefs</i>	Pointer to an array of pointers to the filter coefficient vectors. These are the complex-valued vectors containing the filter coefficients. The dimension of <i>ppSrcDstCoefs</i> is <i>[numSegments][len]</i> .
<i>numSegments</i>	The number of filter segments (L) ($0 < numSegments < 256$).

<i>len</i>	Number of elements contained in each input and output vector ($0 < len \leq 4097$).
<i>scaleFactorCoef</i>	Fixed scale factor for filter coefficients ($0 \leq scaleFactor < 32$). Typically, the same scale factor is used in both <code>ippsCoefUpdateAECNLMS</code> and <code>ippsFilterAECNLMS</code> functions. The filter coefficients are assumed to have been multiplied by $2^{scaleFactorCoef}$ prior to the call to this function. The result of the update is saturated if overflow occurs.

Description

The function `ippsCoefUpdateAECNLMS` is declared in the `ippalign.h` file. This function implements [step 8](#) of the above algorithm to update the adaptive filter coefficients. The scale factor is applied only to the additive update term

$$+ \mathbf{M}_n \bullet \mathbf{X}_{n-i}^* \bullet \mathbf{E}_n$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>ppSrcStepSize</i> , <i>ppSrcFilterInput</i> , <i>pSrcError</i> , or <i>ppSrcDstCoefs</i> pointer is NULL.
<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> has an illegal value.
<code>ippStsRangeErr</code>	Indicates an error when <i>pSrcStepSize[i].val</i> < 0, or when <i>numSegments</i> or <i>scaleFactorCoef</i> is out of range.

Step Size Update Primitives

StepSizeUpdateAECNLMS

Computes the adaptive step size.

Syntax

```
IppStatus ippStepSizeUpdateAECNLMS_32s(const Ipp32s* pSrcInputPSD,
    Ipp32s muQ31, IppAECScaled32s maxStepSize, Ipp32s minInputPSD,
    IppAECScaled32s* pDstStepSize, int len);
```

Parameters

<i>pSrcInputPSD</i>	Pointer to real-valued vector containing the input power spectrum estimate.
<i>muQ31</i>	Real-valued Q31 scalar ($0.0_{Q31} \leq muQ31 < 1.0_{Q31}$).
<i>mu</i>	Real-valued scalar ($0.0 \leq mu < 1.0$).
<i>maxStepSize</i>	Left justified scaled integer representing the maximum step size allowed (usually this is equal to $mu / minInputPSD > 0$).
<i>minInputPSD</i>	Minimum value of input PSD for which the step size update should occur ($0 < minInputPSD$).
<i>len</i>	Number of elements contained in input and output vectors ($0 < len \leq 4097$).
<i>pDstStepSize</i>	Pointer to the left justified scaled integer output vector.

Description

The function `ippStepSizeUpdateAECNLMS` is declared in the `ippalign.h` file. This function implements [step 5](#) of the above algorithm to compute the adaptive step size.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcInputPSD</i> or <i>pDstStepSize</i> pointer is NULL.

<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> has an illegal value.
<code>ippStsRangeErr</code>	Indicates an error when <i>pSrcInputPSD[i]</i> < 0, or when <i>mu</i> , <i>mu31</i> has an illegal value, or <i>minInputPSD</i> <= 0, or <i>maxStepSize.val</i> <= 0, or <i>maxStepSize</i> <= 0, or <i>maxStepSize</i> > 1.

AEC Controller Primitives

ControllerGetSizeAEC

Returns the size of the AEC controller state structure.

Syntax

```
IppStatus ippControllerGetSizeAEC_32s(int* pDstSize);
```

Parameters

pDstSize Pointer to variable that will hold the size in bytes of the state.

Description

The function `ippControllerGetSizeAEC` is declared in the `ippalign.h` file. This function returns the size in bytes of the AEC controller state structure, so that the correct amount of memory can be allocated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDstSize</i> pointer is NULL.

ControllerInitAEC

Initializes the AEC controller state structure.

Syntax

```
IppStatus ippsControllerInitAEC_32s(const IppAECNLMSPParam* pSrcParams,  
    IppAECtrlState* pDstState);
```

Parameters

<i>pSrcParams</i>	Pointer to AEC parameters structure.
<i>pDstState</i>	Pointer to the AEC controller state structure.

Description

The function `ippsControllerInitAEC` is declared in the `ippalign.h` file. This function initializes the AEC controller state structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcParams</i> or <i>pSrcDstState</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>pSrcParams->numSamples</i> or <i>pSrcParams->sampleRate</i> is out of range (see Data Structures for description of range).



NOTE. State memory address stored in *pDstState* must be aligned to 32-bit word boundary.

ControllerUpdateAEC

Implements the energy-based AEC controller.

Syntax

```
IppStatus ippControllerUpdateAEC_32s(const IppAECNLMSParam* pSrcParams,
    IppAECtrlState* pSrcDstState, Ipp32s* pDstMuQ31,
    Ipp32s* pDstAECOutGainQ30, Ipp32s* pDstSpeakerGainQ30);
```

Parameters

<i>pSrcParams</i>	Pointer to AEC parameters structure.
<i>pSrcDstState</i>	Pointer to AEC controller state structure.
<i>pDstMuQ31</i>	Pointer to real-valued Q31 scalar ($0.0_{Q31} \leq pDstMuQ31 < 1.0_{Q31}$).
<i>pDstMu</i>	Pointer to real-valued scalar ($0.0 \leq pDstMu < 1.0$).
<i>pDstAECOutGainQ30</i>	Pointer to real-valued Q30 scalar ($0.0_{Q30} \leq pDstAECOutGainQ30 \leq 1.0_{Q30}$).
<i>pDstAECOutGain</i>	Pointer to real-valued scalar ($0.0 \leq pDstAECOutGain \leq 1.0$).
<i>pDstSpeakerGainQ30</i>	Pointer to real-valued Q30 scalar ($0.0_{Q30} \leq pDstSpeakerGainQ30 \leq 1.0_{Q30}$).
<i>pDstSpeakerGain</i>	Pointer to real-valued scalar ($0.0 \leq pDstSpeakerGain \leq 1.0$).

Description

The function `ippControllerUpdateAEC` is declared in the `ippalign.h` file. This function implements the energy-based AEC controller described in [Algorithm Description](#) section. It sets the fixed adaptation step size (*mu*), the AEC output gain, and the loudspeaker gain based on various system parameters. For example, during “double-talk” (that is, the condition where a person is speaking into the microphone while audio is playing through the loudspeaker), the adaptation step size is reduced.



NOTE. The `ippControllerUpdateAEC` primitive is intended to provide basic AEC control functionality. However, best performance will always be obtained by designing the controller for the specific hardware and physical device geometry in any given application. The purpose of this controller is to provide baseline AEC performance during the prototyping stage. It is expected that this controller will be replaced with a custom controller in a final product. Limitations of this controller include the following:

- It does not maintain adaptive filter convergence for double-talk durations of more than about 2 seconds. This case rarely occurs during normal conversation.
- It does not adapt when the loudspeaker energy is constant for more than a few hundred milliseconds. This case rarely occurs in practice. However, it may have the surprising side-effect that playing wideband stationary noise through the loudspeaker does not cause the adaptive filter to converge rapidly (as would be expected when no controller is present).
- It assumes that there is roughly a 6 dB drop between the loudspeaker and microphone. This is highly dependent on the physical layout and directionality of the loudspeaker and microphone as well as the analog playback gain. Consequently, an AEC which uses this controller may not perform well with an external amplifier at high volume levels.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcParams</code> , <code>pSrcDstState</code> , <code>pDstMuQ31</code> , <code>pDstAECOutGainQ30</code> , <code>pDstSpeakerGainQ30</code> , <code>pDstMu</code> , <code>pDstAECOutGain</code> , or <code>pDstSpeakerGain</code> pointer is NULL.

Voice Activity Detector

This section describes the Intel IPP functions that can be combined to construct a Voice Activity Detector (VAD). The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The VAD consists of parameter estimation and speech modeling heuristics.

IPP VAD primitives support the following features:

- Peak picking
- Periodicity
- Zero crossing rate

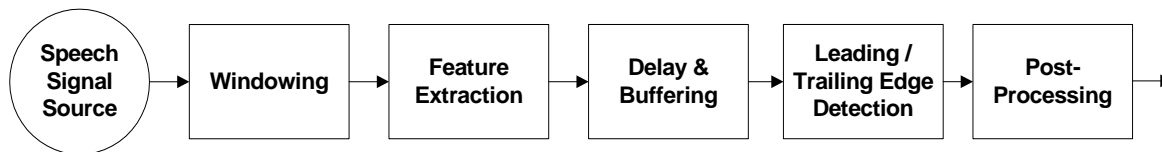
The use of these functions is demonstrated in Intel IPP Speech Recognition Samples. See *Audio Processing Sample* downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Voice Activity Detector Architecture

A typical voice activity detector is illustrated below. The speech signal is windowed and separated into (possibly overlapping) frames. Feature extraction is then performed. Feature extraction may include one or more of the following: zero crossing rate calculation, energy calculation, segmental SNR estimation, periodicity detection, and sample or sub-band histogram measurement. Measurements are buffered so that speech onset and ending can be distinguished from other spurious events. A delay of 400 milliseconds is typical. Speech leading and trailing edge detection is carried out based on heuristic rules. Post-processing (e.g., median filtering) may be applied in low-delay implementations to eliminate false detections.

Figure 8-13 Major blocks in a Voice Activity Detector



Voice Activity Detection Primitives

FindPeaks

Identifies peaks in the input vector.

Syntax

```
IppStatus ippsFindPeaks_32s8u(const Ipp32s* pSrc, Ipp8u* pDstPeaks, int len,  
    int searchSize, int movingAvgSize);  
IppStatus ippsFindPeaks_32f8u(const Ipp32f* pSrc, Ipp8u* pDstPeaks, int len,  
    int searchSize, int movingAvgSize);
```

Parameters

<i>pSrc</i>	Pointer to an input vector.
<i>pDstPeaks</i>	Pointer to the output vector containing a one in positions corresponding to a peak in the input vector and zeros elsewhere.
<i>len</i>	Number of elements contained in the input and output vectors ($0 < len < 65536$).
<i>searchSize</i>	Number of elements on either side to consider when picking peak ($0 < searchSize < 128$).
<i>movingAvgSize</i>	Number of elements on either side to include in moving average window that is applied before peak picking ($0 \leq movingAvgSize < 128$).

Description

The function `ippsFindPeaks` is declared in the `ippsr.h` file. This function identifies peaks in the input vector, places a one in the output vector at the locations of the peaks, and places a zero elsewhere.

A peak is defined as a point $pSrc[i]$ such that $pSrc[i-L] < pSrc[i-L+1] < \dots < pSrc[i-1] < pSrc[i] > \dots > pSrc[i+L-1] > pSrc[i+L]$, where $searchSize$ is L .

If $movingAvgSize$ is greater than 0, then the source vector is smoothed via moving average before peaks are selected.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDstPeaks</i> pointer is NULL.
<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> is out of range.
<code>ippStsSizeErr</code>	Indicates an error when <i>SearchSize</i> or <i>movingAvgSize</i> is out of range.

PeriodicityLSPE

Computes the periodicity of the input speech frame.

Syntax

```
IppStatus ippsPeriodicityLSPE_16s(const Ipp16s* pSrc, int len, Ipp16s*
    pPeriodicityQ15, int* period, int maxPeriod, int minPeriod);
IppStatus ippsPeriodicityLSPE_32f(const Ipp32f* pSrc, int len, Ipp32f*
    pPeriodicity, int* period, int maxPeriod, int minPeriod);
```

Parameters

<i>pSrc</i>	Pointer to an input speech vector.
<i>len</i>	Number of elements contained in the input vector subject to $6 < \text{len} \leq \min(16 * \text{minPeriod}, 1024)$
<i>pPeriodicityQ15</i>	Pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling ($0.0_{Q15} \leq pPeriodicityQ15 \leq 1.0_{Q15}$).
<i>pPeriodicity</i>	Pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling ($0.0 \leq pPeriodicity \leq 1.0$).
<i>period</i>	The period (in samples) that minimizes the LSPE cost function.
<i>maxPeriod</i>	Maximum period to search ($\text{minPeriod} < \text{maxPeriod} < \text{len}$).
<i>minPeriod</i>	Minimum period to search ($6 \leq \text{minPeriod} < \text{maxPeriod}$).

Description

The function `ippsPeriodicityLSPE` is declared in the `ippsr.h` file. This function computes the periodicity of the input speech frame. The periodicity is calculated according the least squares periodicity estimate (LSPE) algorithm defined in [Tuc92]. The periodicity search is designed for speech signals with sample rate between 2000-24000 Hz and may not perform well on music or other sources. If only periodicity (voicing) is required, it is more efficient to downsample input speech before calling this primitive. For example, at 2000 Hz sample rate, the settings `minPeriod=10`, `maxPeriod=20`, and `len=64`, provide adequate periodicity for some applications.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsLengthErr</code>	Indicates an error when <code>len</code> is out of range.
<code>ippStsRangeErr</code>	Indicates an error when <code>maxPeriod</code> or <code>minPeriod</code> is out of range.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pPeriodicityQ15</code> pointer is NULL.

Periodicity

Computes the periodicity of the input block.

Syntax

```
IppStatus ippsPeriodicity_32s16s(const Ipp32s* pSrc, int len, Ipp16s*  
    pPeriodicityQ15, int* period, int maxPeriod, int minPeriod);  
IppStatus ippsPeriodicity_32f (const Ipp32f* pSrc, int len, Ipp32f*  
    pPeriodicity, int* period, int maxPeriod, int minPeriod);
```

Parameters

<code>pSrc</code>	Pointer to an input vector containing non-negative entries.
<code>len</code>	Number of elements contained in the input vector ($0 < len \leq 4096$).
<code>pPeriodicityQ15</code>	Pointer to the Q15 format value corresponding to the normalized sum of the largest harmonic sampling ($0.0_{Q15} \leq pPeriodicityQ15 \leq 1.0_{Q15}$).

<i>pPeriodicity</i>	Pointer to the value corresponding to the normalized sum of the largest harmonic sampling ($0.0 \leq pPeriodicity \leq 1.0$).
<i>period</i>	Pointer to the period (in samples) that provided the maximum-energy harmonic sampling.
<i>maxPeriod</i>	Maximum period to search ($minPeriod < maxPeriod < len$).
<i>minPeriod</i>	Minimum period to search ($0 < minPeriod < maxPeriod$).

Description

The function `ippsPeriodicity` is declared in the `ippsr.h` file. This function computes the periodicity of the input block. In typical applications, the input block is the magnitude-squared of the discrete Fourier transform of windowed speech. The periodicity is defined as the periodic sampling of the input block that preserves the most energy.

$$\max_{k, T_0} \sum_n x(k + nT_0) \quad \text{where } 0 < k \leq T_0$$

Bias removal is performed prior to the search to ensure an accurate measurement.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> is out of range.
<code>ippStsRangeErr</code>	Indicates an error when <i>pSrc</i> [k], <i>maxPeriod</i> , or <i>minPeriod</i> is out of range.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>period</i> , or <i>pPeriodicityQ15</i> pointer is NULL.

Speech Coding Functions

9

This chapter describes Intel[®] IPP functions that can be used for implementing speech codecs which follow ITU-T recommendations G.729, G.723.1, G.722.1, G.726 and G.728, as well as GSM-AMR, GSM-FR codecs, Echo Cancellor and Sub-Band ADPCM codec.

When properly built, such speech codecs can be compliant with the bit-exact specifications for published test vectors.

The chapter also includes the logarithmic encoder-decoder functions that perform the μ -law and A-law companding operations.

The organization of this chapter is the following. The introductory part includes description of rounding modes, defines notational conventions, header files and data structures used by Intel IPP speech coding functions. This is succeeded by the major functionality sections, which include:

- [Common Functions](#)
- [G.729 Related Functions](#)
- [G.723.1 Related Functions](#)
- [GSM-AMR Related Functions](#)
- [AMR Wideband Related Functions](#)
- [GSM Full Rate Related Functions](#)
- [G.722.1 Related Functions](#)
- [G.726 Related Functions](#)
- [G.728 Related Functions](#)
- [Echo Cancellor Related Functions](#)
- [G722 Sub-Band ADPCM Speech Codec Related Functions](#)
- [Companding Functions](#)

Each section starts with the table that gives the full list of Intel IPP functions specific for that functional group, followed by detailed description of the respective API.

The use of the Intel IPP speech coding functions is demonstrated in Intel IPP Samples. See *Intel IPP Speech Coding Samples* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Rounding mode

As many speech codecs have to meet the bit-exact requirement, Intel IPP functions described in this chapter use rounding modes that are different from the default rounding mode used in general signal processing functions.

For general signal processing functions, the default rounding mode can be described as “nearest even”, so that the fixed point number $x = N + \alpha$, $0 \leq \alpha < 1$, where N is an integer number, is rounded as given by:

$$\lceil x \rceil = \begin{cases} N, & 0 \leq \alpha < 0.5 \\ N + 1, & 0.5 < \alpha < 1 \\ N, & \alpha = 0.5, N - \text{even} \\ N + 1, & \alpha = 0.5, N - \text{odd} \end{cases}$$

For example, 1.5 will be rounded to 2 and 2.5 to 2.

For functions in this chapter, there are two rounding modes.

The default rounding mode is “clipping”, so that the fractional part of the fixed point number is cut off and the result of rounding is always less than the initial value. Specifically, the fixed point number $x = N + \alpha$, $0 \leq \alpha < 1$, where N is an integer number, is always rounded to N .

For example, -1.3 will be rounded to -2 and 1.7 to 1. No special suffix is added to the names of functions that use this default mode.

Another rounding mode is “nearest right”, in which the fixed point number $x = N + \alpha$, $0 \leq \alpha < 1$, where N is an integer number, is rounded as follows:

$$\lceil x \rceil = \begin{cases} N, & 0 \leq \alpha < 0.5 \\ N + 1, & 0.5 \leq \alpha < 1 \end{cases}$$

For example, 1.5 will be rounded to 2 (same as for “nearest even” mode) but -1.5 to -1 (different from “nearest even” mode where -1.5 is rounded to -2).

The suffix “NR” is added to names of functions that use the “nearest right” rounding mode.

Notational Conventions

This chapter, in addition to conventions used throughout the manual, uses the following notational conventions:

- In the description of function arguments, when an argument refers to a vector, the expression $[n]$ in square brackets that may be given after the explanation of the argument specifies the number of elements (length) of that vector;
- Most of the speech coding functions for their proper execution interpret their integer and integer array arguments as fixed point numbers, which represent real numbers that vary in their specific ranges. Notation Q_n used in the argument description means that this argument values are used in integer calculations inside the function as real numbers equal to the integer value multiplied by 2^{-n} (where “ n ” is called a scale factor).

For example, if an argument value is described as “4096 in Q_{12} ”, then it is interpreted as the real number 1.0; the value described as “15565 with scale factor 14” represents the real number 0.95; an argument described as “ Q_{14} in $[0, 1]$ ” must be passed as an integer value in the range $[0, 16386]$.

Definitions

This section identifies the header files required for using the Intel IPP speech coding API described later in this chapter. For the definition of bit rate specifiers, refer to [Data Structures](#) section.

The header files `ippdefs.h` and `ippsc.h` must be included in order to link against any of the speech coding primitives, as shown in the following example code:

```
#include "ippdefs.h"
#include "ippsc.h"

int main()
{
    ...
    /* call GSM-AMR IPP functions */
    ippsLevinsonDurbin_GSMAMR(pSrcAutoCorr, pSrcDstLpc);
    ...
}
```

Data Structures

Some of the speech coding functions use a bit rate parameter of the enumerated type `IppSpchBitRate`. The set contains one specifier for each of the supported bit rates, as given in the structure below:

```
typedef enum {
    IPP_SPCHBR_4750
    IPP_SPCHBR_5150
    IPP_SPCHBR_5300
    IPP_SPCHBR_5900
    IPP_SPCHBR_6300
    IPP_SPCHBR_6600
    IPP_SPCHBR_6700
    IPP_SPCHBR_7400
    IPP_SPCHBR_7950
    IPP_SPCHBR_8850
    IPP_SPCHBR_9600
    IPP_SPCHBR_10200
    IPP_SPCHBR_12200
    IPP_SPCHBR_12650
    IPP_SPCHBR_12800
    IPP_SPCHBR_14250
    IPP_SPCHBR_15850
    IPP_SPCHBR_16000
    IPP_SPCHBR_18250
    IPP_SPCHBR_19850
    IPP_SPCHBR_23050
    IPP_SPCHBR_23850
    IPP_SPCHBR_24000
    IPP_SPCHBR_32000
    IPP_SPCHBR_40000
    IPP_SPCHBR_DTX
} IppSpchBitRate;
```

The specifiers `IPP_SPCHBR_4750`, `IPP_SPCHBR_5150`, `IPP_SPCHBR_5900`, `IPP_SPCHBR_6700`, `IPP_SPCHBR_7400`, `IPP_SPCHBR_7950`, `IPP_SPCHBR_10200` and `IPP_SPCHBR_12200` are used in GSM-AMR functions and correspond to 4.75, 5.15, 5.9, 6.7, 7.4, 7.95, 10.2 and 12.2 Kbits/s transmitting rates, respectively.

The specifiers `IPP_SPCHBR_5300` and `IPP_SPCHBR_6300` are used in G.723.1 codec and correspond to 5.3 (low) and 6.3 (high) Kbits/s transmitting rates.

The specifiers `IPP_SPCHBR_9600`, `IPP_SPCHBR_12800` and `IPP_SPCHBR_16000` are used in G.728 codec and correspond to 9.6, 12.8 and 16 Kbits/s transmitting rates.

The specifiers IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000 and IPP_SPCHBR_40000 are used in G.726 codec and correspond to 16, 24, 32 and 40 Kbits/s transmitting rates, respectively.

The specifiers IPP_SPCHBR_6600, IPP_SPCHBR_8850, IPP_SPCHBR_12650, IPP_SPCHBR_14250, IPP_SPCHBR_15850, IPP_SPCHBR_18250, IPP_SPCHBR_19850, IPP_SPCHBR_23050, and IPP_SPCHBR_23850 are used in AMR WB codec and correspond to 6.60, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 Kbits/s transmitting rates, respectively.

Common Functions

This section describes common functions used in different speech codecs. The list of these functions is given in the following table:

Table 9-1 Intel IPP Common Speech Coding Functions

Function Base Name	Operation
ConvPartial	Performs linear convolution of 1D signals.
Mul_NR	Multiplies the elements of two vectors.
MulC_NR	Multiplies each element of a vector by a constant value.
MulPowerC_NR	Performs power weighting for each element of a vector.
AutoScale	Scales by the maximal elements.
DotProdAutoScale	Computes the dot product of two vectors using the automatic scaling.
InvSqrt	Computes inverse square root of vector elements.
AutoCorr	Calculates autocorrelation of a vector.
AutoCorrLagMax	Estimates the maximum auto-correlation of a vector.
AutoCorr_NormE	Estimates normal auto-correlation of a vector.
CrossCorr	Estimates the cross-correlation of two vectors.
CrossCorrLagMax	Estimates the maximum cross-correlation between two vectors.
SynthesisFilter	Computes the speech signal by filtering the input speech through the synthesis filter $1/A(z)$.

ConvPartial

Performs linear convolution of 1D signals.

Syntax

```
IppStatus ippsConvPartial_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsConvPartial_16s32s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp32s* pDst, int len);

IppStatus ippsConvPartial_NR_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
```

Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the source and destination vectors.
<i>scaleFactor</i>	Scale factor used for output data scaling.

Description

The function `ippsConvPartial_16s32s` is declared in the `ippsc.h` file. This function computes the convolution of the vectors *pSrc1* and *pSrc2* as:

$$pDst[i] = \sum_{j=0}^i pSrc1[i] \cdot pSrc2[i-j] , \quad i = 0, ..len-1 .$$

The function `ippsConvPartial_16s_Sfs` computes the convolution of the vectors *pSrc1* and *pSrc2* and scales output data as given by:

$$pDst[i] = 2^{-scaleFactor} \cdot \sum_{j=0}^i pSrc1[i] \cdot pSrc2[i-j] , \quad i = 0, ..len-1$$

Computed results are clipped.

The function `ippsConvPartial_NR_16s` performs the same operation as the function `ippsConvPartial_16s32s`, but uses NR rounding and the default scale factor equal to 15.

Note that the output results produced by the partial convolution function are the same as the first `len` results of the general signal processing convolution function [ippsConv_16s_Sfs](#), if the difference due to rounding is not taken into account.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc1</code> , <code>pSrc2</code> , or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.
<code>ippStsScaleRangeErr</code>	Indicates an error when <code>scaleFactor</code> is negative.

InterpolateC_NR

Computes the weighted sum of two vectors

Syntax

```
IppStatus ippsInterpolateC_NR_16s(const Ipp16s* pSrc1, Ipp16s val1, int
    val1ScaleFactor, const Ipp16s* pSrc2, Ipp16s val2, int val2ScaleFactor,
    Ipp16s* pDst, int len);
```

Parameters

<code>pSrc1</code>	Pointer to the first source vector
<code>val1</code>	First factor
<code>val1ScaleFactor</code>	First factor scale
<code>pSrc2</code>	Pointer to the second source vector
<code>val2</code>	Second factor
<code>val2ScaleFactor</code>	Second factor scale
<code>pDst</code>	Pointer to the destination vector
<code>len</code>	Length of the vectors

Description

This function computes the weighted sum of two vectors as

$$pDst[i] = (2^{val1ScaleFactor} \cdot val1 \cdot pSrc1[i] + 2^{val2ScaleFactor} \cdot val2 \cdot pSrc2[i]) \cdot 2^{-16}, \quad i = 0, \dots, len - 1$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc1</code> , <code>pSrc2</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.
<code>ippStsSizeErr</code>	Indicates an error when <code>ScaleFactor1</code> or <code>ScaleFactor2</code> is less than zero.

Mul_NR

Multiplies the elements of two vectors.

Syntax

```
IppStatus ippsMul_NR_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_NR_16s_ISfs (const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
    int scaleFactor);
```

Parameters

<code>pSrc1</code> , <code>pSrc2</code>	Pointers to the source vectors to be multiplied.
<code>pDst</code>	Pointer to the destination vector.
<code>pSrc</code>	Pointer to the vector to be multiplied by the elements of <code>pSrcDst</code> (for in-place operation).
<code>pSrcDst</code>	Pointer to the source and destination vector (for in-place operation).
<code>len</code>	Number of elements in each vector.

scaleFactor Scale factor for output data scaling.

Description

The function `ippsMul_NR` is declared in the `ippsc.h` file. This function multiplies the first source vector *pSrc1* by the second source vector *pSrc2* element-wise, and stores results in *pDst*.

The in-place function flavor multiplies the vector *pSrc* by the vector *pSrcDst* element-wise, and stores results in *pSrcDst*.

Both function flavors scale the multiplication results in accordance with the *scaleFactor* value (see [“Integer Scaling”](#) in Chapter 2). If output values exceed the data range, the results are saturated.

Functions `ippsMul_NR` perform “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , <i>pSrc</i> , <i>pSrcDst</i> , or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is less than 0.

MulC_NR

*Multiplies each element of a vector
by a constant value.*

Syntax

```
IppStatus ippsMulC_NR_16s_Sfs (Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len,
                               int scaleFactor);
IppStatus ippsMulC_NR_16s_ISfs (Ipp16s val, Ipp16s* pSrcDst, int len, int
                                scaleFactor);
```

Parameters

<i>val</i>	The scalar value used to multiply to each element of the source vector.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor for output data scaling.

Description

The function `ippsMulC_NR` is declared in the `ipps.h` file. This function multiplies the source vector *pSrc* by the scalar value *val*, and stores the result in *pDst*.

The in-place function flavor multiplies the vector *pSrcDst* by *val*, and stores the result in *pSrcDst*.

Both function flavors scale the multiplication results in accordance with the *scaleFactor* value (see [“Integer Scaling”](#) in Chapter 2). If output values exceed the data range, the results are saturated.

Functions `ippsMulC_NR` perform “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is less than 0.

MulPowerC_NR

*Performs power weighting
for each element of a vector.*

Syntax

```
IppStatus ippsMulPowerC_NR_16s_Sfs (const Ipp16s* pSrc, Ipp16s val, Ipp16s*
    pDst, int len, int scaleFactor);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>val</i>	The scalar value to be multiplied to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in each vector.
<i>scaleFactor</i>	Scale factor for output data scaling.

Description

The function `ippsMulPowerC_NR` is declared in the `ippsc.h` file. This function multiplies each element of the source vector *pSrc* by a power of a scalar value, *val*, and stores the results in *pDst*. The calculation is performed as:

$$pDst[i] = val^i \cdot pSrc[i], \quad 0 \leq i < len$$

The function scales the multiplication results in accordance with the *scaleFactor* value (see [“Integer Scaling”](#) in Chapter 2). If output values exceed the data range, the results are saturated.

The function `ippsMulPowerC_NR` performs “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is less than 0.

AutoScale

Scales by the maximal elements.

Syntax

```
IppStatus ippsAutoScale_16s (const Ipp16s* pSrc, Ipp16s* pDst, int len,
                             int* pScale);
IppStatus ippsAutoScale_16s_I(Ipp16s* pSrcDst, int len, int* pScale);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in each vector.
<i>pScale</i>	Pointer to the input/output scaling factor.

Description

The function `ippsAutoScale` is declared in the `ippsc.h` file. This function scales the input vector as follows:

$$pDst[i] = 2^{scaleFactor} \cdot pSrc[i], \quad i = 0, \dots, len-1,$$

where $scaleFactor = normMax - pScale[0]$,

and $normMax$ is calculated so that the maximal absolute value of the vector can be normalized.

The value $pScale[0]$ is the input scaling factor. This scaling factor is replaced by $scaleFactor$ after processing.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , <i>pSrcDst</i> , or <i>pScale</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when the scale factor pointed by <i>pScale</i> is less than 0.

DotProdAutoScale

*Computes the dot product of two vectors
using the automatic scaling.*

Syntax

```
IppStatus ippsDotProdAutoScale_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32s* pDp, int* pSfs);
```

Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>len</i>	Number of elements in each vector.
<i>pDp</i>	Pointer to the output result.
<i>pSfs</i>	Pointer to return the scaling factor.

Description

The function `ippsDotProdAutoScale` is declared in the `ippsc.h` file. This function computes the dot product of two vectors and automatically scales it during calculation, adjusting the scale factor so as to eliminate possible overflow in the process:

$$pDp = 2^{\text{scaleFactor}} \cdot \sum_{i=0}^{\text{len}-1} pSrc1[i] \cdot pSrc2[i]$$

The final scaling factor is returned via *pSfs*.

Vectors *pSrc1* and *pSrc2* must have the same length, *len*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , <i>pDp</i> , or <i>pSfs</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

InvSqrt

Computes inverse square root of vector elements.

Syntax

```
IppStatus ippsInvSqrt_32s_I (Ipp32s* pSrcDst, int len );
```

Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector.
<i>len</i>	Number of elements in the vector.

Description

The function `ippsInvSqrt` is declared in the `ippsc.h` file. This function computes the inverse square roots as:

$$pDst[i] = \frac{2^{31}}{\sqrt{pSrc[i]}}, \quad i = 0, \dots, len-1,$$

using the approximation table. The input and output vectors are scaled by 30 bits.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

AutoCorr

Calculates autocorrelation of a vector.

Syntax

```
IppStatus ippsAutoCorr_16s32s (const Ipp16s* pSrc, int srcLen, Ipp32s* pDst,
    int dstLen );
```

Parameters

<i>pSrc</i>	Pointer to the first source vector [<i>srcLen</i>].
<i>srcLen</i>	Length of the source vector.
<i>pDst</i>	Pointer to the destination vector [<i>dstLen</i>].
<i>dstLen</i>	Length of the destination vector (the number of autocorrelation values to calculate).

Description

The function `ippsAutoCorr` is declared in the `ippsc.h` file. This function calculates autocorrelation of the input vector as follows:

$$pDst[n] = \sum_{i=n}^{srcLen-1} pSrc[i-n] \cdot pSrc[i], \quad n = 0, \dots, dstLen-1.$$

and stores the result in *pDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>srcLen</i> or <i>dstLen</i> is less or equal to 0.

AutoCorrLagMax

Estimates the maximum auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorrLagMax_Inv_16s (const Ipp16s* pSrc, int len, int
    lowerLag, int upperLag, Ipp32s* pMax, int* pMaxLag);
IppStatus ippsAutoCorrLagMax_Fwd_16s (const Ipp16s* pSrc, int len, int
    lowerLag, int upperLag, Ipp32s* pMax, int* pMaxLag);
IppStatus ippsAutoCorrLagMax_32f (const Ipp32f* pSrc, int len, int
    lowerLag, int upperLag, Ipp32f* pMax, int* pMaxLag);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [<i>-upperLag</i> , <i>len</i>].
<i>len</i>	Length of autocorrelation.
<i>lowerLag</i>	Lower input lag value.
<i>upperLag</i>	Upper input lag value.
<i>pMax</i>	Pointer to the output maximum of the correlation.
<i>pMaxLag</i>	Pointer to the output lag which holds the maximum of the correlation.

Description

These functions are declared in the `ipps.h` file. The functions `ippsAutoCorrLagMax_Inv` and `ippsAutoCorrLagMax_32f` find the maximum autocorrelation within the given lag range as:

$$pMax = \max_n \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i-n] , \text{ lowerLag} \leq n \leq \text{upperLag} .$$

The function `ippsAutoCorrLagMax_Fwd` finds the maximum autocorrelation within the given lag range as given by:

$$pMax = \max_n \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n] , \text{ lowerLag} \leq n \leq \text{upperLag} .$$

If several maximums exist, functions return the first of them.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pMax</i> , or <i>pMaxLag</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

AutoCorr_NormE

Estimates normal auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorr_NormE_16s32s (const Ipp16s* pSrc, int len, Ipp32s* pDst,
    int lenDst, int* pNorm);
IppStatus ippsAutoCorr_NormE_NR_16s (const Ipp16s* pSrc, int len, Ipp16s* pDst,
    int lenDst, int* pNorm);
```

Parameters

<i>pSrc</i>	Pointer to the source vector in Q12.
<i>len</i>	Number of elements in the source vector.
<i>pDst</i>	Pointer to the destination vector that stores the estimated auto-correlation results of the source vector.
<i>lenDst</i>	The number of elements in the destination vector.
<i>pNorm</i>	Pointer to the output scale factor.

Description

The function `ippsAutoCorr_NormE` is declared in the `ippsc.h` file. This function calculates the autocorrelation of the source vector *pSrc*. The results are scaled according to the first autocorrelation coefficient (energy) value. Specifically, autocorrelation coefficients are multiplied by the factor 2^{norm} , where $norm \geq 0$ is calculated so as to make the first coefficient normalized:

$$pDst[n] = 2^{norm} \cdot \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n] , \quad 0 \leq n < lenDst , \quad 0 \leq norm$$

where

$$pSrc[i] = \begin{cases} pSrc[i] , & 0 \leq i < len \\ 0 , & otherwise \end{cases}$$

The corresponding scaling factor, *norm*, is returned via *pNorm*.

The function that has the NR suffix performs the “nearest right” rounding (see [Rounding mode](#)).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> or <code>pNorm</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> or <code>lenDst</code> is less than or equal to zero.
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

CrossCorr

Estimates the cross-correlation of two vectors.

Syntax

```
IppStatus ippsCrossCorr_16s32s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp32s* pDst, int scaleFactor);
IppStatus ippsCrossCorr_NormM_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp16s* pDst);
```

Parameters

<code>pSrc1</code>	Pointer to the first source vector.
<code>pSrc2</code>	Pointer to the second source vector.
<code>len</code>	Number of elements in the source and destination vectors.
<code>pDst</code>	Pointer to the destination vector.
<code>scaleFactor</code>	Scale factor of the destination vector.

Description

The function `ippsCrossCorr` is declared in the `ippsc.h` file. This function estimates the cross-correlation between the vector `pSrc1` and the vector `pSrc2` as given by:

$$corr[n] = 2^{scaleFactor} \cdot \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i+n] \quad ,$$

where

$$pSrc2[i] = \begin{cases} pSrc2[i] , & 0 \leq i < len \\ 0 , & otherwise \end{cases}$$

Results are stored in the vector *pDst*. Correlation sums are saturated.
Scaling is performed according to the *scaleFactor* value.

The function that has the `NormM` suffix uses the scale factor that normalizes the absolute maximum of the correlation sums.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , or <i>pDst</i> pointer is NULL.
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is negative.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

CrossCorrLagMax

Estimates the maximum cross-correlation between two vectors.

Syntax

```
IppStatus ippCrossCorrLagMax_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,  
    int len, int lag, Ipp32s* pMax, int* pMaxLag);  
IppStatus ippCrossCorrLagMax_32f64f (const Ipp32f* pSrc1, const Ipp32f* pSrc2,  
    int len, int lag, Ipp64f* pMax, int* pMaxLag);
```

Parameters

<i>pSrc1</i>	Pointer to the first source vector [<i>len</i>].
<i>pSrc2</i>	Pointer to the second source vector [<i>len</i> + <i>lag</i> +1].
<i>len</i>	Length of the cross-correlation.
<i>lag</i>	The maximal lag value.

<i>pMax</i>	Pointer to the output maximum cross-correlation.
<i>pMaxLag</i>	Pointer to the output lag which holds the cross-correlation maximum.

Description

The functions `ippsCrossCorrLagMax` are declared in the `ippsc.h` file. These functions find the maximum of the cross-correlation between two input vectors by the formulae:

$$pMax = \max_{0 \leq n \leq lag} \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i+n]$$

The function `ippsCrossCorrLagMax_16s` returns *pMax* value multiplied by two.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , <i>pMax</i> , or <i>maxLag</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>lag</i> is less than 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.

SynthesisFilter

Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).

Syntax

```
IppStatus ippsSynthesisFilter_NR_16s_Sfs(const Ipp16s* pLPC, const Ipp16s*
    pSrc, Ipp16s* pDst, int len, int scaleFactor, const Ipp16s* pMem);
IppStatus ippsSynthesisFilterLow_NR_16s_ISfs(const Ipp16s* pLPC, Ipp16s*
    pSrcDst, int len, int scaleFactor, const Ipp16s* pMem);
IppStatus ippsSynthesisFilter_NR_16s_ISfs(const Ipp16s* pLPC, Ipp16s*
    pSrcDst, int len, int scaleFactor, const Ipp16s* pMem);
```

Parameters

<i>pLPC</i>	Pointer to the input filter coefficients a_0, a_1, \dots, a_{10} .
-------------	--

<i>pSrc</i>	Pointer to the input vector.
<i>pSrcDst</i>	Pointer to the history input/filtered output vector.
<i>pDst</i>	Pointer to the filtered output.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor for the destination vector.
<i>pMem</i>	Pointer to the memory supplied for filtering. Should be updated outside the function with last 10 values of the destination vector.

Description

The function `ippSsynthesisFilter` is declared in the `ippsc.h` file. This function filters the input signal through the synthesis filter as follows:

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i), n = 0, 1, \dots, len-1.$$

The function flavor `ippSsynthesisFilterLow_NR_16s_ISfs` performs no saturation and assumes that input data are small enough to ensure correct calculation without checking the overflow condition. This mode of operation is indicated by the suffix `Low` in the function name.

All flavors perform “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pLPC</i> , <i>pSrcDst</i> , <i>pDst</i> , or <i>pMem</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is not equal to 12 or 13.
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

G.729 Related Functions

Intel IPP functions described in this section implement building blocks that can be used to create speech codecs compliant with the ITU-T Recommendation G.729 (see [\[ITU729\]](#), [\[ITU729A\]](#), [\[ITU729B\]](#)).

These functions are used in the Intel IPP *G.729 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

The list of these functions is given in the following table:

Table 9-2 Intel IPP G.729 Related Functions

Function Base Name	Operation
Basic Functions	
DotProd_G729	Computes the dot product of two vectors.
Interpolate_G729	Computes the weighted sum of two vectors.
Linear Prediction Analysis Functions	
AutoCorr_G729	Estimates the auto-correlation of a vector.
LevinsonDurbin_G729	Calculates LP coefficients from the autocorrelation coefficients.
LPCToLSP_G729	Converts LP coefficients to LSP coefficients.
LSFToLSP_G729	Converts Line Spectral Frequencies to LSP coefficients.
LSFQuant_G729	Performs quantization of LSF coefficients.
LSFDecode_G729	Decodes quantized LSFs.
LSFDecodeErased_G729	Reconstructs quantized LSFs in case when a frame is erased.
LSPToLPC_G729	Converts LSP coefficients to LP coefficients.
LSPQuant_G729	Quantizes the LSP coefficients.
LSPToLSF_G729	Converts LSP coefficients to LSF coefficients.
LagWindow_G729	Applies 60Hz bandwidth expansion.
Codebook Search Functions	
OpenLoopPitchSearch_G729	Searches for an optimal pitch value.
AdaptiveCodebookSearch_G729	Searches for the integer delay and the fraction delay, and computes the adaptive vector.
DecodeAdaptiveVector_G729	Restores the adaptive codebook vector by interpolating the past excitation.
FixedCodebookSearch_G729	Searches for the fixed codebook vector.

Table 9-2 Intel IPP G.729 Related Functions (continued)

Function Base Name	Operation
<u>ToeplizMatrix_G729</u>	Calculates 616 elements of the Toepliz matrix for the fixed codebook search.
Codebook Gain Functions	
<u>DecodeGain_G729</u>	Decodes the adaptive and fixed-codebook gains.
<u>GainControl_G729</u>	Calculates adaptive gain control.
<u>GainQuant_G729</u>	Quantizes the codebook gain using a two-stage conjugate-structured codebook.
<u>AdaptiveCodebookContribution_G729</u>	Updates target vector for codebook search by subtracting adaptive codebook contribution.
Filter Functions	
<u>ResidualFilter_G729</u>	Implements an inverse LP filter and obtains the residual signal.
<u>SynthesisFilter_G729</u>	Reconstructs the speech signal from LP coefficients and residuals.
<u>LongTermPostFilter_G729</u>	Restores the long-term information from the old speech signal.
<u>ShortTermPostFilter_G729</u>	Restores speech signal from the residuals.
<u>TiltCompensation_G729</u>	Compensates for the tilt in the short-term filter.
<u>HarmonicFilter</u>	Calculates the harmonic filter.
<u>HighPassFilterSize_G729</u>	Returns the G729 high-pass filter size.
<u>HighPassFilterInit_G729</u>	Initializes the high-pass filter.
<u>HighPassFilter_G729</u>	Performs G729 high-pass filtering.
<u>IIR16s_G729</u>	Performs IIR filtering.
<u>PhaseDispersionGetStateSize_G729D</u>	Queries the memory length of the phase dispersion filter.
<u>PhaseDispersionInit_G729D</u>	Initializes the phase dispersion filter memory.
<u>PhaseDispersionUpdate_G729D</u>	Updates the phase dispersion filter state.
<u>PhaseDispersion_G729D</u>	Performs the phase dispersion filtering.
<u>Preemphasize_G729A</u>	Computes pre-emphasis of a post filter.
<u>WinHybridGetStateSize_G729E</u>	Queries the length of the hybrid windowing module memory.
<u>WinHybridInit_G729E</u>	Initializes the hybrid windowing module memory.
<u>WinHybrid_G729E</u>	Applies the hybrid window and computes autocorrelation coefficients.
<u>RandomNoiseExcitation_G729B</u>	Initializes a random vector with a Gaussian distribution.
<u>FilteredExcitation_G729</u>	Computes filtered excitation.

Basic Functions

These functions may be applied in both the encoding and decoding process.

DotProd_G729

Computes the dot product of two vectors.

Syntax

```
IppStatus ippsDotProd_G729A_16s32s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp32s* pDp);
IppStatus ippsDotProd_G729A_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, int
    len, Ipp32f* pDp);
```

Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>len</i>	Number of elements in each vector.
<i>pDp</i>	Pointer to the output result.

Description

The functions `ippsDotProd_G729A` are declared in the `ippsc.h` file. The function `ippsDotProd_G729A_16s32s` computes the dot product of two source vectors *pSrc1* and *pSrc2* as:

$$pDp = 2 \cdot \sum_{i=0}^{len/2} pSrc1[2 \cdot i] \cdot pSrc2[2 \cdot i],$$

while for the `ippsDotProd_G729A_32f` function the result is

$$pDp = \sum_{i=0}^{len/2} pSrc1[2 \cdot i] \cdot pSrc2[2 \cdot i]$$

Vectors *pSrc1* and *pSrc2* must have the same length, *len*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , or <i>pDp</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

Interpolate_G729

Computes the weighted sum of two vectors.

Syntax

```
IppStatus ippInterpolate_G729_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
IppStatus ippInterpolateC_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s val1,
    const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippInterpolateC_NR_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s val1,
    const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippInterpolateC_G729_32f (const Ipp32f* pSrc1, Ipp32f val1, const
    Ipp32f* pSrc2, Ipp32f val2, Ipp32f* pDst, int len);
```

Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>val1</i>	Factor to multiply to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>val2</i>	Factor to multiply to the second source vector.
<i>pDst</i>	Pointer to the destination (interpolated) vector.
<i>len</i>	Number of elements in the vectors.
<i>scaleFactor</i>	Scale factor for the destination vector.

Description

These functions are declared in the `ippsc.h` file. The function `ippsInterpolate_G729_16s` computes the weighted sum as:

$$pDst[i] = (pSrc1[i] + \text{sign}(pSrc1[i])) \gg 1 + (pSrc2[i] + \text{sign}(pSrc2[i])) \gg 1 ,$$

for $i = 0, \dots, len-1$.

Functions `ippsInterpolateC_G729_16s_Sfs` and `ippsInterpolateC_NR_G729_16s_Sfs` both use the same formula to compute the weighted sum as:

$$pDst[i] = (val1 \cdot pSrc1[i] + val2 \cdot pSrc2[i]) \gg scaleFactor , i = 0, \dots, len-1 .$$

However, these functions differ in the rounding mode they use for output results (see [Rounding mode](#)).

The function `ippsInterpolateC_G729_32f` computes the weighted sum as:

$$pDst(i) = val1 \cdot pSrc1(i) + val2 \cdot pSrc2(i) , 0 \leq i < len$$

Return Values

- | | |
|----------------------------------|---|
| <code>ippStsNoErr</code> | Indicates no error. |
| <code>ippStsNullPtrErr</code> | Indicates an error when the <code>pSrc1</code> , <code>pSrc2</code> , or <code>pDst</code> pointer is NULL. |
| <code>ippStsSizeErr</code> | Indicates an error when <code>len</code> is less than or equal to 0. |
| <code>ippStsScaleRangeErr</code> | Indicates an error when <code>scaleFactor</code> is less than 0. |

Linear Prediction Analysis Functions

Functions described in this section implement LSP coding (quantization) and decoding, as well as transformation between LPC, LSP and LSF coefficients.

AutoCorr_G729

Estimates the auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorr_G729B(const Ipp16s* pSrcSpch,
    Ipp16s* pResultAutoCorrExp, Ipp32s* pDstAutoCorr);
```

Parameters

pSrcSpch Pointer to the input speech signal vector [240].
pResultAutoCorrExp Pointer to the exponent of autocorrelation coefficients.
pDstAutoCorr Pointer to the autocorrelation coefficients vector [13].

Description

The function `ippsAutoCorr_G729` is declared in the `ippsc.h` file. This function calculates a set of 11 autocorrelation coefficients and their exponent for the input speech signal. The function is applied to the vector of 240 speech samples, which include 120 samples from past speech frames, 80 samples from the present speech frame, and 40 samples from the future frame. The functionality is as follows.

1. First, apply to speech samples the asymmetric windows given by:

$$w_{1p}(n) = \begin{cases} 0.54 - 0.46 \cos\left(\frac{2n\pi}{399}\right), & n = 0, 1, \dots, 199 \\ \cos\left(\frac{2(n-200)\pi}{159}\right), & n = 200, 201, \dots, 239 \end{cases}$$

2. Next, calculate autocorrelations of the windowed speech samples $s(i)$, $i = 0, 1, \dots, 239$, using the formula:

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k), \quad k = 0, 1, \dots, 11$$

3. Finally, the autocorrelation coefficients are scaled according to the first autocorrelation coefficient (energy) value. Specifically, autocorrelation coefficients are multiplied by the factor 2^{norm} , where $norm \leq 0$ is calculated so as to make the first coefficient normalized. The corresponding scaling factor, $norm$, is returned via *pResultAutoCorrExp*.



NOTE. The function `ippsAutoCorr_G729B` is actually a combination of `ippsMul_NR` and `ippsAutoCorr_NormE` functions. The following code details the correspondence.

```
{
    short sig_win[240];
    ippsMul_NR_16s_Sfs(pSrcSpch, window, sig_win, 240, 15);
    ippsAutoCorr_NormE_16s32s(sig_win, 240, pDstAutoCorr, 11,
        &pResultAutoCorrExp);
}
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcSpch</i> , <i>pResultAutoCorrExp</i> or <i>pDstAutoCorr</i> pointer is NULL.

LevinsonDurbin_G729

Calculates LP coefficients from the autocorrelation coefficients.

Syntax

```
IppStatus ippsLevinsonDurbin_G729_32s16s(const Ipp32s* pSrcAutoCorr, int order,
    Ipp16s* pDstLPC, Ipp16s* pDstRc, Ipp16s* pResultResidualEnergy);
IppStatus ippsLevinsonDurbin_G729_32f(const Ipp32f* pSrcAutoCorr, int order,
    Ipp32f* pDstLpc, Ipp32f* pDstRc, Ipp32f* pResultResidualEnergy);
IppStatus ippsLevinsonDurbin_G729B(const Ipp32s* pSrcAutoCorr, Ipp16s*
    pDstLPC, Ipp16s* pDstRC, Ipp16s* pResultResidualEnergy);
```


Parameters

<i>order</i>	The LP order.
<i>pSrcAutoCorr</i>	Pointer to the autocorrelation coefficients vector [<i>order</i> +1].
<i>pDstLPC</i>	Pointer to the output LP coefficients [<i>order</i> +1].
<i>pDstRC</i>	Pointer to the output reflection coefficients vector [<i>order</i>].
<i>pResultResidualEnergy</i>	Pointer to the residual energy.

Description

The functions `ippsLevinsonDurbin_G729` and `ippsLevinsonDurbin_G729B` are declared in the `ippsc.h` file. These functions calculate Linear Prediction (LP) coefficients of the LP filter with the given order from the autocorrelation coefficients, using Levinson-Durbin algorithm. The function `ippsLevinsonDurbin_G729` uses the parameter *order*, while the function `ippsLevinsonDurbin_G729B` operates for the default *order* = 10.

To obtain LP coefficients a_i , $i = 1, 2, \dots, order$, the following set of equations is to be solved:

$$\sum_{i=1}^{order} a_i \times r(|i-k|) = -r(k) \text{ , } k = 1, 2, \dots, order \text{ .}$$

The functions perform the following steps:

1. Levinson-Durbin algorithm is applied to solve the above set of equations. This algorithm uses the following recursion.

$$\begin{aligned}
 E^{[0]} &= r(0) \\
 \text{for } i &= 1 \text{ to } order \\
 a_0^{[i-1]} &= 1 \\
 k_i &= - \left[\sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j) \right] / E^{[i-1]} \\
 a_i^{[i]} &= k_i \\
 \text{for } j &= 1 \text{ to } i-1
 \end{aligned}$$

```

         $a_j^{[i]} = a_j^{[i-1]} + k_i \times a_{i-j}^{[i-1]}$ 
    end
     $E^{[i]} = E^{[i-1]} - k_i^2 E^{[i-1]}$ 
end

```

2. Set E as the output residual energy and k_i as reflection coefficients.
3. If the LPC filter used in this algorithm is unstable, that is some $|k_i|$ is very close to 1.0 during recursion, the elements of input vectors $pSrcDstLPC$ and $pSrcDstRC$ RC coefficients are not changed and the residual energy is set to 0.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcAutoCorr</i> , <i>pDstRC</i> , <i>pResultResidualEnergy</i> , or <i>pDstLPC</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>order</i> is less than or equal to 0, or when <i>order</i> is greater than 30.

LPCToLSP_G729

Converts LP coefficients to LSP coefficients.

Syntax

```

IppStatus ippsLPCToLSP_G729_16s(const Ipp16s* pSrcLPC, const Ipp16s*
    pSrcPrevLSP, Ipp16s* pDstLSP);
IppStatus ippsLPCToLSP_G729A_16s(const Ipp16s* pSrcLPC, const Ipp16s*
    pSrcPrevLSP, Ipp16s* pDstLSP);
IppStatus ippsLPCToLSP_G729A_32f(const Ipp32f* pSrcLPC, const Ipp32f*
    pSrcPrevLsp, Ipp32f* pDstLSP);
IppStatus ippsLPCToLSP_G729_32f(const Ipp32f* pSrcLPC, const Ipp32f*
    pSrcPrevLsp, Ipp32f* pDstLSP);

```

Parameters

<i>pSrcLPC</i>	Pointer to the LP coefficients vector [11], in Q12 for 16s data.
----------------	--

<i>pSrcPrevLSP</i>	Pointer to the previous LSP coefficients vector [10], in Q15 for 16s data.
<i>pDstLSP</i>	Pointer to the computed LSP coefficients vector [10].

Description

These functions are declared in the `ippsc.h` file. They convert 10th order LP coefficients to LSP coefficients.

The first function `ippsLPCToLSP_G729` is designed for G.729/B codec while the second function `ippsLPCToLSP_G729A` is designed for G.729A codec.

Functions perform the following steps:

1. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the following recursive relations:

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i), \quad i = 0, 1, \dots, 4,$$

where $f_1(0) = f_2(0) = 1.0$.

2. Use Chebyshev polynomials to evaluate $F_1(z)$ and $F_2(z)$. The Chebyshev polynomials are given by.

$$C_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + f_1(5)/2$$

$$C_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + f_2(5)/2$$

For G.729/B, evaluate $F_1(z)$ and $F_2(z)$ at 60 points equally spaced between 0 and π and check for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided four times to track the root.

For G.729A, evaluate $F_1(z)$ and $F_2(z)$ at 50 points equally spaced between 0 and π and check for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided two times to track the root.

3. If all 10 roots needed to determine LSP coefficients are not found, just use the previous set of LSP coefficients.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcLPC</i> , <i>pSrcPrevLSP</i> or <i>pDstLSP</i> pointer is NULL.

LSFToLSP_G729

Converts Line Spectral Frequencies to LSP coefficients.

Syntax

```
IppStatus ippsLSFToLSP_G729_16s (const Ipp16s* pLSF, Ipp16s* pLSP);
```

Parameters

<i>pLSF</i>	Pointer to the LSF vector [10], in Q13 in the range $[0, \pi]$.
<i>pLSP</i>	Pointer to the LSP vector [10], in Q15 in the range $[-1; 1]$.

Description

The function `ippsLSFToLSP_G729` is declared in the `ippsc.h` file. This function converts the Line Spectral Frequencies (LSF) to the LSP coefficients as given by:

$$pLSP[i] = \cos(pLSF[i]), i = 1, \dots, 10.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pLSP</i> or <i>pLSF</i> pointer is NULL.
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

LSFQuant_G729

Performs quantization of LSF coefficients.

Syntax

```
IppStatus ippsLSFQuant_G729_16s (const Ipp16s* pLSF, Ipp16s* pQuantLSFTable,
    Ipp16s* pQuantLSF, Ipp16s* quantIndex);
IppStatus ippsLSFQuant_G729B_16s (const Ipp16s* pLSF, Ipp16s* pQuantLSFTable,
    Ipp16s* pQuantLSF, Ipp16s* quantIndex);
```

```
IppStatus ippsLSFQuant_G729B_32f (const Ipp32f* pLSF, Ipp32f* pQuantLSFTable,
    Ipp32f* pQuantLSF, int* quantIndex);
```

Parameters

<i>pLSF</i>	Pointer to the LSF vector [10], in Q13 in the range $[0, \pi]$ for 16s data.
<i>pQuantLSFTable</i>	Pointer to a matrix of 4 rows and 10 columns, containing previously quantized LSFs.
<i>pQuantLSF</i>	Pointer to the quantized LSF vector, in Q13 in the range $[0, \pi]$ for 16s data.
<i>quantIndex</i>	Pointer to the output combined codebook indices <i>L0</i> , <i>L1</i> , <i>L2</i> , and <i>L3</i> .

Description

These functions are declared in the `ippsc.h` file.

The function `ippsLSFQuant_G729_16s` quantizes the difference between the input LSF coefficients and previously quantized LSF coefficients, using the switched Moving Average (MA) predictor.

Quantization is performed using a two-stage Vector Quantizer (VQ): 10-dimensional VQ using codebook **L1** (128 entries), and 10-bit split VQ (5 dimensions each) on two codebooks, **L2** and **L3** (32 entries each).

The function `ippsLSFQuant_G729B_16s` quantizes the LSF coefficients using a two-stage split VQ, (in 5 and 4 bits). The second order MA predictor used in the SID-LPC quantization procedure is calculated as a linear interpolation of the first and second MA predictors, with weight values 0.6 and 0.4, respectively. The first stage of quantization is the same as that in G.729, but only part of the quantization table (32 entries) is used. The second stage is different. No splitting is done and also only portion of the second table (16 entries) is used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pLSF</i> , <i>pQuantLSFTable</i> , <i>pQuantLSF</i> , or <i>quantIndex</i> pointer is NULL.
<code>ippStsLSFLow</code>	Indicates a warning that the corresponding filter has low stability.
<code>ippStsLSFHigh</code>	Indicates a warning that the corresponding filter has high stability.
<code>ippStsLSFLowAndHigh</code>	Indicates a warning that the corresponding filter has both low and high stability.

LSFDecode_G729

Decodes quantized LSFs.

Syntax

```
IppStatus ippsLSFDecode_G729_16s (const Ipp16s* quantIndex, Ipp16s*
    pQuantLSFTable, Ipp16s* pQuantLSF);
IppStatus ippsLSFDecode_G729B_16s (const Ipp16s* quantIndex, Ipp16s*
    pQuantLSFTable, Ipp16s* pQuantLSF);
IppStatus ippsLSFDecode_G729B_32f (const int* quantIndex, Ipp32f*
    pQuantLSFTable, Ipp32f* pDstQLsp);
IppStatus ippsLSFDecode_G729_32f (const int* quantIndex, Ipp32f*
    pQuantLSFTable, Ipp32f* pQuantLSF);
```

Parameters

<i>quantIndex</i>	Pointer to the vector of indices: <i>L0</i> , <i>L1</i> , <i>L2</i> , <i>L3</i> (see Table 8/G.729 in ITU729) or <i>L0</i> , <i>L1</i> , <i>L2</i> (see also ITU729B , section 4.3).
<i>pQuantLSFTable</i>	Pointer to the input/output table [4][10] of 4 previously quantized LSFs vectors, in Q13 for 16s data.
<i>pQuantLSF</i>	Pointer to the quantized LSF output vector [10], in Q13 for 16s data.

Description

These functions are declared in the `ippsc.h` file.

The function `ippsLSFDecode_G729` retrieves the quantized LSF coefficients from the quantization table and indices.

The function `ippsLSFDecode_G729B` retrieves the quantized LSF coefficients for the SID frame. Only *L0*, *L1*, *L2* indices are used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>quantIndex</i> , <i>pQuantLSFTable</i> , or <i>pQuantLSF</i> pointer is NULL.
<code>ippStsLSFLow</code>	Indicates a warning of only low stability of LSF.
<code>ippStsLSFHigh</code>	Indicates a warning of only high stability of LSF.

`ippStsLSFLowAndHigh` Indicates a warning of both low and high stability of LSF.

LSFDecodeErased_G729

*Reconstructs quantized LSFs in case
when a frame is erased.*

Syntax

```
IppStatus ippsLSFDecodeErased_G729_16s (Ipp16s maIndex, Ipp16s* pQuantLSFTable,
    Ipp16s* pQuantLSF);
IppStatus ippsLSFDecodeErased_G729_32f (int maIndex, Ipp32f* pQuantLSFTable,
    const Ipp32f* pSrcPrevLSF);
```

Parameters

<i>maIndex</i>	Switched MA predictor (<i>L0</i>) .
<i>pQuantLSFTable</i>	Pointer to the input/output table [4][10] of 4 previously quantized LSFs vectors, in Q12 for 16s data.
<i>pQuantLSF</i>	Pointer to the quantized LSF output vector [10], in Q12 for 16s data.

Description

The function `ippsLSFDecodeErased_G729` is declared in the `ippsc.h` file. This function retrieves the quantized LSF coefficients through the previously decoded switched MA-predictor index (*L0*).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pQuantLSFTable</i> or <i>pQuantLSF</i> pointer is NULL.
<code>ippStsOutOfRangeErr</code>	Indicates an error when $maIndex < 0$ or $maIndex \geq MA_NP$.

LSPToLPC_G729

Converts LSP coefficients to LP coefficients.

Syntax

```
IppStatus ippsLSPToLPC_G729_16s(const Ipp16s* pSrcLSP, Ipp16s* pDstLPC);
IppStatus ippsLSPToLPC_G729_32f(const Ipp32f* pSrcLsp, Ipp32f* pDstLpc);
```

Parameters

pSrcLSP Pointer to the LSP coefficients vector [10], in Q15 for 16s data.
pDstLPC Pointer to the LP coefficients vector [11], in Q12 for 16s data.

Description

The function `ippsLSPToLPC_G729` is declared in the `ipps.h` file. This function converts a set of 10th order LSP coefficients to LP coefficients.

It performs the following steps:

1. Calculates the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursive relations:

for $i = 1$ to 5

$$f_1(i) = -2q_{2i-1} \times f_1(i-1) + 2f_1(i-2)$$

for $j = i-1$ down to 1

$$f_1^{[i]}(j) = f_1^{[i-1]}(j) - 2q_{2i-1} \times f_1^{[i-1]}(j-1) + f_1^{[i-1]}(j-2)$$

end

end

Here the initial values are set to $f_1(0) = 1$ and $f_1(-1) = 0$.

The coefficients $f_2(i)$ are computed similarly by replacing q_{2i-1} by q_{2i} .

2. $F_1(z)$ and $F_2(z)$ are then multiplied by $1 + z^{-1}$ and $1 - z^{-1}$ respectively to obtain $F_1'(z)$ and $F_2'(z)$ as given by:

$$f_1'(i) = f_1(i) + f_1(i-1), i = 1, 2, \dots, 5$$

$$f_2'(i) = f_2(i) - f_2(i-1), i = 1, 2, \dots, 5$$

3. Finally, the function computes the LP coefficients from $f_1'(i)$ and $f_2'(i)$ as:

$$a_i = \begin{cases} 0.5 \times f_1'(i) + 0.5 \times f_2'(i) , & i = 1, 2, \dots, 5 \\ 0.5 \times f_1'(11-i) - 0.5 \times f_2'(11-i) , & i = 6, 7, \dots, 10 \end{cases}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcLSP</i> or <i>pDstLPC</i> pointer is NULL.

LSPQuant_G729

Quantizes the LSP coefficients.

Syntax

```
IppStatus ippLSPQuant_G729_16s(const Ipp16s* pSrcLSP, Ipp16s*
    pSrcDstPrevFreq, Ipp16s* pDstQLSP, Ipp16s* pDstQLSPIndex);
IppStatus ippLSPQuant_G729E_32f(const Ipp32f* pSrcLSP, Ipp32f*
    pSrcDstPrevFreq, Ipp32f* pDstQLSF, Ipp32f* pDstQLSP, int* pDstQLSPIndex);
```

Parameters

<i>pSrcLSP</i>	Pointer to the LSP coefficients vector [10], in Q15 for 16s data.
<i>pSrcDstPrevFreq</i>	Pointer to the four previous and updated quantized frequencies vector [40], in Q13 for 16s data. Elements 0 to 9 are for the newest frame and elements 30 to 39 are for the oldest frame.
<i>pDstQLSF</i>	Pointer to the LSF coefficients vector [10].
<i>pDstQLSP</i>	Pointer to the LSP coefficients vector [10], in Q15 for 16s data.
<i>pDstQLSPIndex</i>	Pointer to the combined codebook indices <i>L0</i> , <i>L1</i> , <i>L2</i> and <i>L3</i> , of length 2. Its first element represents indices <i>L0</i> and <i>L1</i> in the following combination: <i>L1</i> occupies the first 7 bits starting from LSB, whereas <i>L0</i> resides in the next 1 bit adjacent to <i>L1</i> . The second element represents indices <i>L2</i> and <i>L3</i> in the following combination: <i>L3</i> occupies the first 5 bits starting from LSB, and <i>L2</i> resides in the next 5 bits adjacent to <i>L3</i> .

Description

The functions `ippLSPQuant_G729` are declared in the `ippsc.h` file. These functions obtain the quantized LSP coefficients and the codebook indices. They perform the following operations:

1. Converts the LSP coefficients α_i to LSF coefficients ω_i in the normalized frequency domain $[0, \pi]$ as:

$$\omega_i = \arccos(\alpha_i) \text{ , } i = 1, \dots, 10$$

2. Uses a switched 4th order MA prediction to predict the LSF coefficients of the current frame. The difference between the computed and predicted coefficients is quantized using a two-stage vector quantizer. The first stage is a 10-dimensional VQ using codebook **L1** with 128 entries (7 bits). The second stage is a 10 bit VQ which was implemented as a split VQ using two 5-dimensional codebooks, **L2** and **L3**, containing 32 entries (5 bits) each. To explain the quantization process, it is convenient to first describe the decoding process. Each coefficient is obtained from the sum of two codebooks:

$$\hat{l}_i = \begin{cases} \mathbf{L1}_i(L1) + \mathbf{L2}_i(L2) \text{ , } i = 1, \dots, 5 \\ \mathbf{L1}_i(L1) + \mathbf{L3}_{i-5}(L3) \text{ , } i = 6, \dots, 10 \end{cases}$$

where $L1$, $L2$ and $L3$ are the codebook indices.

3. The vector to be quantized for the current frame m is obtained from:

$$l_i = \left[\omega_i^{(m)} - \sum_{k=1}^4 \hat{p}_{i,k} \hat{l}_i^{(m-k)} \right] / \left(1 - \sum_{k=1}^4 \hat{p}_{i,k} \right) \text{ , } i = 1, \dots, 10 \text{ ,}$$

where $\hat{p}_{i,k}$ are the coefficients of the switched MA predictor. The first codebook **L1** is searched and the entry $L1$ that minimizes the (unweighted) mean-squared error is selected. This is followed by a search of the second codebook **L2**, which defines the lower part of the second stage. To avoid sharp resonances in the quantized LP synthesis filter, the partial vector \hat{l}_i , $i = 1, \dots, 5$, is rearranged such that adjacent coefficients have a minimum distance of J . The rearrangement procedure is shown below:

```

for i = 2,...,10
    if  $\hat{l}_{i-1} > \hat{l}_i - J$ 
         $\hat{l}_{i-1} = (\hat{l}_i + \hat{l}_{i-1} - J) / 2$ 
         $\hat{l}_i = (\hat{l}_i + \hat{l}_{i-1} + J) / 2$ 
    end
end
end
```

where J is 0.0012 for the first pass. Using the selected first stage vector $L1$ and the lower part of the second stage $L2$, the higher part of the second stage is searched from the codebook $L3$. Again, the rearrangement procedure is used to guarantee a minimum distance of 0.0012. The resulting vector \hat{l}_i , $i = 1, \dots, 10$, is rearranged to guarantee a minimum distance of 0.0006. The quantized LSF coefficients $\hat{\omega}_i^{(m)}$ for the current frame m are obtained from the weighted sum of previous quantizer outputs $\hat{l}_i^{(m-k)}$, and the current quantizer output $\hat{l}_i^{(m)}$:

$$\hat{\omega}_i^{(m)} = \left(1 - \sum_{k=1}^4 \hat{p}_{i,k} \right) \hat{l}_i^{(m)} + \sum_{k=1}^4 \hat{p}_{i,k} \hat{l}_i^{(m-k)}, \quad i = 1, \dots, 10$$

There are two MA predictors. Which MA predictor to use is defined by a single bit $L0$. It is determined by the one that minimizes the weighted mean-squared error:

$$E_{lsf} = \sum_{i=1}^{10} w_i (\omega_i - \hat{\omega}_i)^2$$

The weights w_i are made adaptive as a function of the unquantized LSF coefficients,

$$w_1 = \begin{cases} 1.0 & , \text{ if } \omega_2 - 0.04\pi - 1 > 0 \\ 10(\omega_2 - 0.04\pi - 1)^2 + 1, & \text{ otherwise} \end{cases}$$

$$w_i = \begin{cases} 1.0 & , \text{ if } \omega_{i+1} - \omega_{i-1} - 1 > 0 \\ 10(\omega_{i+1} - \omega_{i-1} - 1)^2 + 1, & \text{ otherwise} \end{cases}, \quad 2 \leq i \leq 9$$

$$w_{10} = \begin{cases} 1.0 & , \text{ if } -\omega_9 + 0.92\pi - 1 > 0 \\ 10(-\omega_9 + 0.92\pi - 1)^2 + 1, & \text{ otherwise} \end{cases}$$

In addition, the weights w_5 and w_6 are multiplied by 1.2 each.

4. To avoid sharp resonance in the LP filter, the rearrangement procedure is applied twice to the resulting vector \hat{l}_i with $J = 0.0012$ and 0.0006 sequentially.

5. After computing $\hat{\omega}_i$, the corresponding filter is checked for stability. This is done as follows:

- 1) Order the coefficients $\hat{\omega}_i$ in increasing value;
- 2) If $\hat{\omega}_i < 0.005$ then $\hat{\omega}_i = 0.005$;
- 3) If $\hat{\omega}_{i+1} - \hat{\omega}_i < 0.0391$ then $\hat{\omega}_{i+1} = \hat{\omega}_i + 0.0391$, $i = 1, \dots, 9$;
- 4) If $\hat{\omega}_{10} > 3.315$ then $\hat{\omega}_{10} = 3.315$.

6. Convert the quantized LSF vectors to LSP vectors.



NOTE. The function `ippsLSPQuant` is actually a combination of [ippsLSPToLSF](#), [ippsLSFQuant](#), and [ippsLSFToLSP](#) functions. The following code details the correspondence.

```
IppStatus ippsLSPQuant_G729_16s(const Ipp16s* pSrcLSP, Ipp16s*
pSrcDstPrevFreq, Ipp16s* pDstQLSP, Ipp16s* pDstQLSPIndex) {
    __ALIGN(8) short lsf[LP_ORDER];
    __ALIGN(8) short lsp_q[LP_ORDER];
    short q_index[4];
    IppStatus sts=ippsLSPToLSF_G729_16s(pSrcLSP, lsf);
    if (sts != ippsStsNoErr) return sts;
    sts = ippsLSFQuant_G729_16s(lsf, pSrcDstPrevFreq, lsp_q, q_index );
    if(sts != ippsStsNoErr) return sts;
    pDstQLSPIndex[0] = (q_index[0]<<G729_NC0_B) | q_index[1];
    pDstQLSPIndex[1] = (q_index[2]<<G729_NC1_B) | q_index[3];
    ippsLSFToLSP_G729_16s(lsp_q, pDstQLSP)
    return ippsStsNoErr;
}
```

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <i>pSrcLSP</i> , <i>pSrcDstPrevFreq</i> , <i>pDstQLSP</i> , <i>pDstQLSF</i> or <i>pDstQLSPIndex</i> pointer is NULL.

LSPToLSF_G729

Converts LSP coefficients to LSF coefficients.

Syntax

```
IppStatus ippsLSPToLSF_Norm_G729_16s (const Ipp16s* pLSP, Ipp16s* pLSF);
IppStatus ippsLSPToLSF_G729_16s (const Ipp16s* pLSP, Ipp16s* pLSF);
```

Parameters

<i>pLSP</i>	Pointer to the LSP vector, in Q15 in the range [-1:1].
<i>pLSF</i>	Pointer to the LSF vector. Values must be scaled by 13 bits in the range [0: π] (for <code>ippsLSPToLSF_G729_16s</code> function) or by 15 bits in the range [0:0.5] (for <code>ippsLSPToLSF_Norm_G729_16s</code> function).

Description

These functions are declared in the `ippsc.h` file.

The function `ippsLSPToLSF_Norm_G729` converts the LSP coefficients to LSF coefficients as follows:

$$pLSF[i] = 2^{scaleFactor} \cdot \arccos(pLSP[i]) \ , 0 \leq i < 10 \ .$$

The scale factor is chosen such that the first LSF coefficient is normalized to the interval [0:0.5] by multiplying to $1/\pi$.

The function `ippsLSPToLSF_G729` retains the result within the interval [0: π]. Both functions use the same approximation table to calculate the inverse cosine.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pLSP</i> or <i>pLSF</i> pointer is NULL.

LagWindow_G729

Applies 60Hz bandwidth expansion.

Syntax

```
IppStatus ippsLagWindow_G729_32s_I (Ipp32s* pSrcDst, int len);
```

Parameters

<i>pSrcDst</i>	Pointer to the autocorrelation vector.
<i>len</i>	Number of elements in the vector.

Description

The function `ippSLagWindow_G729` is declared in the `ippsc.h` file. This function applies a 60Hz bandwidth expansion to the autocorrelation vector as follows:

$$r[0] = 1.0001 \cdot r[0]$$

$$r[i] = w_{lag}[i] \cdot r[i], \quad i = 0, \dots, len-1,$$

where

$$w_{lag}[i] = \exp\left(-\frac{1}{2} \cdot \left(2\pi f_0 \cdot \frac{i}{f_s}\right)^2\right)$$

and

$$f_0 = 60\text{Hz}, \quad f_s = 8000\text{Hz}.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsRangeErr</code>	Indicates an error when <code>len > 12</code> .

Codebook Search Functions

These functions perform open loop pitch estimation using the adaptive codebook, and search in ACELP excitation (fixed) codebook for optimal signs and positions of the pulses.

OpenLoopPitchSearch_G729

Searches for an optimal pitch value.

Syntax

```
IppStatus ippsOpenLoopPitchSearch_G729_16s(const Ipp16s* pSrc, Ipp16s*
    bestLag);
IppStatus ippsOpenLoopPitchSearch_G729A_16s(const Ipp16s* pSrc, Ipp16s*
    bestLag);
IppStatus ippsOpenLoopPitchSearch_G729A_32f(const Ipp32f* pSrc, Ipp16s*
    bestLag);
```

Parameters

pSrc Pointer to the perceptually weighted speech signal vector [170].
bestLag Pointer to the open-loop pitch search result.

Description

These functions are declared in the `ippsc.h` file. They use an open-loop method to get the pitch value. To reduce the complexity of the search for the best adaptive-codebook delay, the search range is limited around a candidate delay, which is obtained from maximizing the correlation coefficients of the perceptually weighted speech.

The function `ippsOpenLoopPitchSearch_G729A` is designed for G.729A codec and implements the following algorithm:

1. Overflow and underflow check:

$$nSumTemp = \sum_{j=-71}^{40} sw(2j-1) \times sw(2j-1);$$

if $nSumTemp \geq 2^{30}$, overflow happens, $sw'(i) = sw(i) > 3$, $i = -143, \dots, 79$;

else if $nSumTemp < 2^{19}$, underflow happens, $sw'(i) = sw(i) < 3$, $i = -143, \dots, 79$;

else $sw'(i) = sw(i)$, $i = -143, \dots, 79$.

2. Compute the correlation:

$$R(k) = \sum_{n=0}^{39} sw'(2n) \times sw'(2n-k), k = 20, \dots, 143$$

3. In the range [20, 39], find the maximum of $R(k)$, and optimal k_1 . In the range [40, 79], find the maximum of $R(k)$, and optimal k_2 . In the range [80, 143], among the even k , find the maximum of $R(k)$, and suboptimal k_3 . Let $k_3' = k_3$.

4. if $R(k_3' + 1) > R(k_3)$, then $k_3 = k_3' + 1$.

5. if $R(k_3' - 1) > R(k_3)$, then $k_3 = k_3' - 1$.

6. For the three optimal k_i , $i = 1, 2, 3$, normalize the three peak values of correlation:

$$R'(k_i) = \frac{R(k_i)}{\sqrt{\sum_{n=0}^{39} sw'(2n - k_i) \times sw'(2n - k_i)}}$$

7. Post process $R'(k_1)$ and $R'(k_2)$:

$$\begin{aligned} & \text{if } (|2 \times k_2 - k_3| < 5) \{ \\ & \quad R'(k_2) = R'(k_2) + 0.25 \times R'(k_3); \\ & \} \\ & \text{if } (|3 \times k_2 - k_3| < 7) \{ \\ & \quad R'(k_2) = R'(k_2) + 0.25 \times R'(k_3); \\ & \} \\ & \text{if } (|2 \times k_1 - k_2| < 5) \{ \\ & \quad R'(k_1) = R'(k_1) + 0.2 \times R'(k_2); \\ & \} \\ & \text{if } (|3 \times k_1 - k_2| < 7) \{ \\ & \quad R'(k_1) = R'(k_1) + 0.2 \times R'(k_2); \\ & \} \end{aligned}$$

8. Find the optimal delay k_{opt} from k_1 to k_3 , which satisfies

$$R'(k_{opt}) = \max \{R'(k_i), i = 1, \dots, 3\}.$$

The function `ippsOpenLoopPitchSearch_G729` is designed for G.729/B codec and implements the following algorithm:

1. Overflow and underflow check:

$$nSumTemp = \sum_{i=-143}^{79} sw(i) \times sw(i)$$

if $nSumTemp \geq 2^{30}$, overflow happens, $sw'(i) = sw(i) >> 3, i = -143, \dots, 79$;

else if $nSumTemp < 2^{19}$, underflow happens, $sw'(i) = sw(i) << 3, i = -143, \dots, 79$;

else $sw'(i) = sw(i), i = -143, \dots, 79$.

2. Compute the correlation:

$$R(k) = \sum_{i=0}^{79} sw'(i) \times sw'(i-k), k = 20, \dots, 143$$

3. In the range $[20, 39]$, find the maximum of $R(k)$, and optimal $k = k_1$. In the range $[40, 79]$, find the maximum of $R(k)$, and optimal $k = k_2$. In the range $[80, 143]$, find the maximum of $R(k)$, and optimal $k = k_3$.

4. For the three optimal $k_i, i = 1, 2, 3$, normalize the three peak values of correlation:

$$R'(k_i) = \frac{R(k_i)}{\sqrt{\sum_{n=0}^{79} sw'(n-k_i) \times sw'(n-k_i)}}$$

2. Find the optimal delay k_{opt} from k_1 to k_3 , according to the following relations:

$$k_{opt} = k_1, \quad R'(k_{opt}) = R'(k_1)$$

$$\text{if } (R'(k_2) \geq 0.85 R'(k_{opt})), k_{opt} = k_2, \quad R'(k_{opt}) = R'(k_2)$$

$$\text{if } (R'(k_3) \geq 0.85R'(k_{opt})) , k_{opt} = k_3 , \quad R'(k_{opt}) = R'(k_3)$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>bestLag</code> pointer is NULL.

AdaptiveCodebookSearch_G729

Searches for the integer delay and the fraction delay, and computes the adaptive vector.

Syntax

```
IppStatus ippAdaptiveCodebookSearch_G729_16s(Ipp16s valOpenDelay, const
    Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
    pSrcDstPrevExcitation, Ipp16s* pDstDelay, Ipp16s* pDstAdptVector, Ipp16s
    subFrame);

IppStatus ippAdaptiveCodebookSearch_G729A_16s(Ipp16s valOpenDelay, const
    Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
    pSrcDstPrevExcitation, Ipp16s* pDstDelay, Ipp16s* pDstAdptVector, Ipp16s
    subFrame);

IppStatus ippAdaptiveCodebookSearch_G729D_16s (Ipp16s valOpenDelay, const
    Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
    pSrcDstPrevExcitation, Ipp16s* pDstDelay, Ipp16s subFrame);
```

Parameters

<code>valOpenDelay</code>	Open-loop delay, in the range [18,145].
<code>pSrcAdptTarget</code>	Pointer to the target signal for adaptive-codebook search vector [40].
<code>pSrcImpulseResponse</code>	Pointer to the impulse response of weighted synthesis filter vector [40], in Q12.
<code>pSrcDstPrevExcitation</code>	Pointer to the previous and updated excitation vector [194].
<code>pDstDelay</code>	Pointer to the integer delay and fraction delay vector [2].
<code>pDstAdptVector</code>	Pointer to the adaptive vector [40].
<code>subFrame</code>	Subframe number, either 0 or 1.

Description

These functions are declared in the `ippsc.h` file. They search for the integer delay T and the fraction delay t , using the target signal and the past filtered excitation, and compute the adaptive vector. They are applied in subframes.

The function `ippsAdaptiveCodebookSearch_G729A` is designed for G.729A codec and implements the following algorithm:

1. Search for the optimal delay k , which maximizes the following expression:

$$\begin{aligned}
 R_N(k) &= \sum_{n=0}^{39} x(n)y_k(n) = \sum_{n=0}^{39} x(n) \sum_{i=0}^n u_k(i)h(n-i) = \sum_{i=0}^{39} \sum_{n=i}^{39} u_k(i)x(n)h(n-i) \\
 &= \sum_{n=0}^{39} u_k(n)x_d(n), \quad k = t_{min}, \dots, t_{max} \\
 x_d(n) &= \sum_{i=n}^{39} x(i)h(i-n), \quad n = 0, 1, \dots, 39.
 \end{aligned}$$

Denote the optimal delay k as T .

2. If the search result T is less than 85 in the first subframe or it is in the second subframe, search for the fraction delay by maximizing the expression below:

$$\begin{aligned}
 R_{Nt}(k) &= \sum_{n=0}^{39} x_d(n)u_{kt}(n), \quad t = -1, 0, 1, \quad k = T \\
 u_{kt}(n) &= \sum_{i=0}^9 u_k(n-k_p-i)b_{30}(t_p+3i) + \sum_{i=0}^9 u_k(n-k_p+1+i)b_{30}(3-t_p+3i), \\
 n &= 0, 1, \dots, 39
 \end{aligned}$$

$$k_p = \begin{cases} T, & t = 0, -1 \\ T+1, & t = 1 \end{cases}, \quad t_p = \begin{cases} -t, & t = 0, -1 \\ 2, & t = 1 \end{cases}$$

3. If the fraction result is t , then the new adaptive vector is computed as:

$$v(n) = u_{k_t}(n) , n = 0,1,...,39 .$$

The function **ippsAdaptiveCodebookSearch_G729** is designed for G.729/B codec and implements the following algorithm:

1. Search for the integer delay k which maximizes the following expression:

$$R(k) = \frac{\sum_{n=0}^{39} x(n)y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n)y_k(n)}}$$

where y_k is the past filter excitation at delay k . Denote the optimal k as T .

2. If the search result T is less than 85 in the first subframe or it is in the second subframe, search for the fraction delay t by maximizing the expression below:

$$R_t(T) = \sum_{i=0}^3 R(T-i)b_{12}(t+3i) + \sum_{i=0}^3 R(T+1+i)b_{12}(3-t+3i) , t = 0,1,2 .$$

3. Finally, obtain the adaptive vector by interpolating the past excitation.

The interpolation method is the same as the method used to get the new adaptive vector in the **ippsAdaptiveCodebookSearch_G729A** function.

The function **ippsAdaptiveCodebookSearch_G729D** differs from **ippsAdaptiveCodebookSearch_G729** in that the number of lags in the second subframe has been reduced from 32 to 16.

Return Values

ippsStsNoErr	Indicates no error.
ippsStsNullPtrErr	Indicates an error when the <i>pSrcAdptTarget</i> , <i>pSrcImpulseResponse</i> , <i>pSrcDstPrevExcitation</i> , <i>pDstDelay</i> , or <i>pDstAdptVector</i> pointer is NULL.
ippsStsRangeErr	Indicates an error when <i>valOpenDelay</i> is not in the range [18, 145], or <i>subFrame</i> is not 0 or 1.

DecodeAdaptiveVector_G729

Restores the adaptive codebook vector by interpolating the past excitation.

Syntax

```
IppStatus ippsDecodeAdaptiveVector_G729_16s(const Ipp16s* pSrcDelay, Ipp16s*
    pSrcDstPrevExcitation, Ipp16s* pDstAdptVector);
IppStatus ippsDecodeAdaptiveVector_G729_16s_I(const Ipp16s* pSrcDelay, Ipp16s*
    pSrcDstPrevExcitation);
IppStatus ippsDecodeAdaptiveVector_G729_32f_I(const Ipp32s* pSrcDelay, Ipp32f*
    pSrcDstPrevExcitation);
```

Parameters

pSrcDelay Pointer to the integer and fraction delay for each subframe.

pSrcDstPrevExcitation Pointer to the past and updated excitations vector [194].

pDstAdptVector Pointer to the adaptive codebook vector [40].

Description

The function `ippsDecodeAdaptiveVector_G729` is declared in the `ippsc.h` file. This function restores the adaptive codebook vector by interpolating the past excitation. It performs the following steps:

1. Decodes the integer part and fractional part of the pitch delay T and t .
2. Interpolates the past excitation to get the current adaptive excitation as:

$$v(n) = \sum_{i=0}^9 u(n-T-i) \times b_{30}(t+3i) + \sum_{i=0}^9 u(n-T+1+i) \times b_{30}(3-t+3i), n = 0,1,...,39$$

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pSrcDelay*, *pSrcDstPrevExcitation*, or *pDstAdptVector* pointer is NULL.

`ippStsRangeErr` Indicates an error when `pSrcDelay[0]` is not in the range [18, 145], or `pSrcDelay[1]` is not -1, 0 or 1.

FixedCodebookSearch_G729

Searches for the fixed codebook vector.

Syntax

```

IppStatus ippsFixedCodebookSearch_G729_16s(const Ipp16s* pSrcFixedCorr,
    Ipp16s* pSrcDstMatrix, Ipp16s* pDstFixedVector, Ipp16s* pDstFixedIndex,
    Ipp16s* pSearchTimes, Ipp16s subFrame);

IppStatus ippsFixedCodebookSearch_G729_32s16s(const Ipp16s* pSrcFixedCorr,
    Ipp32s* pSrcDstMatrix, Ipp16s* pDstFixedVector, Ipp16s* pDstFixedIndex,
    Ipp16s* pSearchTimes, Ipp16s subFrame);

IppStatus ippsFixedCodebookSearch_G729A_16s(const Ipp16s* pSrcFixedCorr,
    Ipp16s* pSrcDstMatrix, Ipp16s* pDstFixedVector, Ipp16s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729A_32s16s(const Ipp16s* pSrcFixedCorr,
    Ipp32s* pSrcDstMatrix, Ipp16s* pDstFixedVector, Ipp16s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729E_16s(int mode, const Ipp16s*
    pSrcFixedTarget, const Ipp16s* pSrcLtpResidual, const Ipp16s*
    pSrcImpulseResponse, Ipp16s* pDstFixedVector, Ipp16s* pDstFltFixedVector,
    Ipp16s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729D_16s(const Ipp16s* pSrcFixedCorr, const
    Ipp16s* pSrcImpulseResponse, Ipp16s* pSrcDstMatrix, Ipp16s* pDstFixedVector,
    Ipp16s* pDstFltFixedVector, Ipp16s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729_32f (const Ipp32f* pSrcFixedCorr,
    Ipp32f* pSrcDstMatrix, Ipp32f* pDstFixedVector, Ipp32s* pDstFixedIndex,
    Ipp32s* pSearchTimes, Ipp32s subFrame);

IppStatus ippsFixedCodebookSearch_G729A_32f (const Ipp32f* pSrcFixedCorr,
    Ipp32f* pSrcDstMatrix, Ipp32f* pDstFixedVector, Ipp32s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729D_32f (Ipp32f* pSrcDstFixedCorr, Ipp32f*
    pSrcDstMatrix, const Ipp32f* pSrcImpulseResponse, Ipp32f* pDstFixedVector,
    Ipp32f* pDstFltFixedVector, Ipp32s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729E_32f (int mode, Ipp32f*
    pSrcDstFixedCorr, const Ipp32f* pSrcLtpResidual, const Ipp32f*
    pSrcImpulseResponse, Ipp32f* pDstFixedVector, Ipp32f* pDstFltFixedVector,
    Ipp32s* pDstFixedIndex);

```

Parameters

<i>mode</i>	Indicates forward (<i>mode</i> = 0) or backward (<i>mode</i> = 1) LP analysis mode.
<i>pSrcFixedCorr</i>	Pointer to the correlation signal of fixed-codebook search vector [40], in Q13.
<i>pSrcDstMatrix</i>	Pointer to the Toeplitz matrix of length 616 (elements of <code>Ipp16s</code> type in Q9 or <code>Ipp32s</code> type in Q25).
<i>pSrcFixedTarget</i>	Pointer to the input updated target speech vector.
<i>pSrcLtpResidual</i>	Pointer to the input residual after long term prediction vector.
<i>pSrcImpulseResponse</i>	Pointer to the LP synthesis filter [40].
<i>pDstFixedVector</i>	Pointer to the fixed-codebook vector [40], in Q13.
<i>pDstFltFixedVector</i>	Pointer to the output filtered fixed-codebook vector.
<i>pDstFixedIndex</i>	Pointer to the fixed-codebook index. The first element is the sign codeword <i>S</i> and the second element is the position codeword <i>C</i> . Index length is the following: 2 - for all G729, all G729A, and G729D_32f flavors 4 - for all G729E flavors 1 - for G729D_16s flavor.
<i>pSearchTimes</i>	Pointer to the maximum search time for the remaining subframes.
<i>subFrame</i>	Subframe number, 0 or 1.

Description

These functions are declared in the `ippsc.h` file. They search for the fixed-codebook vector and corresponding vector index, respectively. They are applied in subframes.

The first function `ippsFixedCodebookSearch_G729` is designed for G.729/B codec and uses the nested-loop search approach. The Toeplitz matrix is updated in this function. This function performs the following steps:

1. First, the 616 elements in the Toeplitz matrix are updated as:

$$\Phi(i, j) = \text{sign}[d(i)]\text{sign}[d(j)]\Phi(i, j)$$

where $\Phi(i, j)$ is the element of the Toeplitz matrix, and $d(n)$ is the correlation signal.

2. Then, a threshold thr_3 is calculated using:

$$thr_3 = av_3 + 0.4(max_3 - av_3)$$

where max_3 and av_3 are the absolute maximum and absolute average of the first three pulses in $d(n)$.

3. Next, search for the fixed-codebook vector to minimize the term C^2 / E , and obtain the vector index, where C is the correlation given by:

$$C = |\bar{d}(m_0)| + |\bar{d}(m_1)| + |\bar{d}(m_2)| + |\bar{d}(m_3)| \quad (9-1)$$

and E is the energy obtained by:

$$\begin{aligned} E = & 2(\Phi'(m_0, m_0) + \Phi'(m_1, m_1) + \Phi'(m_0, m_1) \\ & + \Phi'(m_2, m_2) + \Phi'(m_0, m_2) + \Phi'(m_1, m_2) \\ & + \Phi'(m_3, m_3) + \Phi'(m_0, m_3) + \Phi'(m_1, m_3) + \Phi'(m_2, m_3)) \end{aligned} \quad (9-2)$$

The search procedure is a kind of 4-loop search, and the last loop is entered only if the pre-computed correlation has exceeded the threshold thr_3 . Also, the maximum number of times the loop can be entered is fixed to 180 so that a low percentage of the codebook is searched.

The sign codeword of the index, S , is obtained as follows.

$$S = s_0 + 2s_1 + 4s_2 + 8s_3 \quad (9-3)$$

The position codeword of the index, C , is computed as:

$$C = (m_0 / 5) + 8(m_1 / 5) + 64(m_2 / 5) + 512(2(m_3 / 5) + jx) \quad , \quad (9-4)$$

where $jx = 0$ if $m_3 = 3, 8, \dots, 38$, and $jx = 1$ if $m_3 = 4, 9, \dots, 39$.

4. Finally, the fixed-codebook vector is calculated.

The function `ippsFixedCodebookSearch_G729A` is designed for G.729A codec and uses the iterative depth-first, tree search approach.

The functionality is as follows:

1. First, calculate 616 elements of the new Toepliz matrix, $\Phi'(i, j)$, from the old Toepliz matrix Φ , using the formula:

$$\Phi'(i, j) = \text{sign}[\bar{d}(i)]\text{sign}[\bar{d}(j)]\Phi(i, j) \quad ,$$

where $\Phi(i, j)$ is the element of the Toepliz matrix, and $d(n)$ is the correlation signal.

2. Next, search for the fixed-codebook vector to minimize the term C^2 / E , and obtain the vector index, where C is the correlation given by (9-1), and E is the energy obtained as (9-2).

The search is an iterative depth-first, tree search approach.

The sign codeword of the index, S , is computed according to (9-3) and the position codeword of the index, C , is obtained as given by (9-4).

3. Finally, the fixed-codebook vector is calculated.

Both `ippsFixedCodebookSearch_G729` and `ippsFixedCodebookSearch_G729A` functions search four signed pulses in positions given by following table:

Pulses	Positions
i_0	0,5,10,15,20,25,30,35
i_1	1,6,11,16,21,26,31,36
i_2	2,7,12,17,22,27,32,37
i_3	3,8,13,18,23,28,33,38 4,9,14,19,24,29,34,39

The function `ippsFixedCodebookSearch_G729E` is designed for G.729E codec and searches ten signed pulses in five tracks in positions shown in the following table:

Pulses	Positions
i_0, i_1	0,5,10,15,20,25,30,35
i_3, i_4	1,6,11,16,21,26,31,36
i_5, i_6	2,7,12,17,22,27,32,37
i_7, i_8	3,8,13,18,23,28,33,38
i_9, i_{10}	4,9,14,19,24,29,34,39

The two pulses for each track may overlap resulting in a single pulse with duplicated amplitude.

The function `ippsFixedCodebookSearch_G729D` is designed for G.729D codec and searches for two signed pulses in two overlapping tracks given in following table:

Pulses	Positions
i_0	1, 3, 6, 8, 11, 13, 16, 18, 21, 23, 26, 28, 31, 33, 36, 38
i_1	0, 1, 2, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 16, 17, 19, 20, 21, 22, 24, 25, 26, 27, 29, 30, 31, 32, 34, 35, 36, 37, 39

Return Value

`ippsStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when any of the pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>subFrame</i> is not 0 or 1 (for integer functions only).

GainCodebookSearch

Finds pitch and fixed gains.

Syntax

```
IppStatus ippsGainCodebookSearch_G729_32f( const Ipp32f* pSrcCorrFactors,
      Ipp32f valPredictedCodebookGain, const int* pSrcCand, Ipp32f* pDstIdxs,
      Ipp32s valTimeFlag);

IppStatus ippsGainCodebookSearch_G729D_32f (const Ipp32f* pSrcCorrFactors,
      Ipp32f valPredictedCodebookGain, const int* pSrcCand, Ipp32f* pDstIdxs,
      Ipp32s valTimeFlag);
```

Parameters

<i>pSrcCorrFactors</i>	Pointer to the correlations.
<i>valPredictedCodebookGain</i>	Predicted codebook gain.
<i>pSrcCand</i>	Pointer to the pre-searched candidates.
<i>pDstIdxs</i>	Pointer to the output indexes.
<i>valTimeFlag</i>	Taming process indicator.

Description

As part of LSP quantization algorithm (refer to [GainQuant_G729](#) for details), these functions operate in floating-point arithmetic and perform selection of the gain vector in the codebook *GA* and *GB* in the range limited by *pSrcCand[0]* and *pSrcCand[1]*. The search algorithm consists of minimizing the following expression:

$$E = pSrcCorrFactors[0] \times G_p^2 + pSrcCorrFactors[2] \times G_c^2 + pSrcCorrFactors[1] \times G_p \\ + pSrcCorrFactors[3] \times G_c + pSrcCorrFactors[4] \times G_p \times G_c$$

where G_p and G_c are optimum pitch and code gains, respectively, in selected domain.

The `ippsGainCodebookSearch_G729_32f` function uses the 7-bit codebook, whereas the function `ippsGainCodebookSearch_G729D_32f` uses the 6-bit codebook.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcCorrFactors</i> , <i>pSrcCand</i> , or <i>pDstIdxs</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>valTimeFlag</i> is not 0 or 1.
<code>ippStsRangeErr</code>	<p><i>For the G729 function flavor:</i></p> <p>Indicates an error when <i>pSrcCand[0]</i> is not in the range [0,4) or <i>pSrcCand[0]</i> is not in the range [0,8).</p> <p><i>For the G729D function flavor:</i></p> <p>Indicates an error when <i>pSrcCand[0]</i> or <i>pSrcCand[1]</i> is not in the range [0,2).</p>

ToeplizMatrix_G729

Calculates elements of the Toepliz matrix for the fixed codebook search.

Syntax

```
IppStatus ippsToeplizMatrix_G729_16s(const Ipp16s* pSrcImpulseResponse,
    Ipp16s* pDstMatrix);
IppStatus ippsToeplizMatrix_G729_16s32s(const Ipp16s* pSrcImpulseResponse,
    Ipp32s* pDstMatrix);
IppStatus ippsToeplizMatrix_G729_32f(const Ipp32f* pSrcImpulseResponse, ,
    Ipp32f* pDstMatrix);
IppStatus ippsToeplizMatrix_G729D_32f(const Ipp32f* pSrcImpulseResponse, ,
    Ipp32f* pDstMatrix);
```

Parameters

pSrcImpulseResponse Pointer to the impulse response vector [40], in Q12 for 16s data.

pDstMatrix Pointer to the elements of Toepliz matrix of length 616,
(of `Ipp16s` type in Q9 or `Ipp32s` type in Q25).

Description

The function `ippsToeplizMatrix_G729` is declared in the `ippsc.h` file. This function calculates 616 elements in Toepliz matrix for the fixed-codebook search. These elements can be expressed as:

$$\Phi(i, j) = \sum_{n=j}^{39} h(n-i) \times h(n-j), \quad 0 \leq i \leq 39, \quad 0 \leq j \leq 39$$

where $h(i)$, $i = 0, 1, \dots, 39$, is the impulse response.

The function stores the calculated 616 elements in *pDstMatrix* in the following order:

1. $\Phi(m_i, m_i)$ ($i = 0, 1, 2, 3$), $3 \times 8 + 16 = 40$ elements; starting location: 0.
 $\Phi(0, 0), \Phi(5, 5), \dots, \Phi(35, 35), \Phi(1, 1), \Phi(6, 6), \dots, \Phi(36, 36),$
 $\Phi(2, 2), \Phi(7, 7), \dots, \Phi(37, 37), \Phi(3, 3), \Phi(8, 8), \dots, \Phi(39, 39).$
2. $\Phi(m_0, m_1)$, $8 \times 8 = 64$ elements; starting location: 40.
 $\Phi(0, 1), \dots, \Phi(0, 36), \Phi(5, 1), \dots, \Phi(5, 36), \Phi(10, 1), \dots, \Phi(35, 36).$
3. $\Phi(m_0, m_2)$, $8 \times 8 = 64$ elements; starting location: 104.
 $\Phi(0, 2), \dots, \Phi(0, 37), \Phi(5, 2), \dots, \Phi(5, 37), \Phi(10, 2), \dots, \Phi(35, 37).$
4. $\Phi(m_0, m_3)$, $8 \times 16 = 128$ elements; starting location: 168.
 $\Phi(0, 3), \dots, \Phi(0, 38), \Phi(5, 3), \dots, \Phi(5, 38), \Phi(10, 3), \dots, \Phi(35, 38).$
 $\Phi(0, 4), \dots, \Phi(0, 39), \Phi(5, 4), \dots, \Phi(5, 39), \Phi(10, 4), \dots, \Phi(35, 39).$
5. $\Phi(m_1, m_2)$, $8 \times 8 = 64$ elements; starting location: 296.
 $\Phi(1, 2), \dots, \Phi(1, 37), \Phi(6, 2), \dots, \Phi(6, 37), \Phi(11, 2), \dots, \Phi(36, 37).$
6. $\Phi(m_1, m_3)$, $8 \times 16 = 128$ elements; starting location: 360.
 $\Phi(1, 3), \dots, \Phi(1, 38), \Phi(6, 3), \dots, \Phi(6, 38), \Phi(11, 3), \dots, \Phi(36, 38).$
 $\Phi(1, 4), \dots, \Phi(1, 39), \Phi(6, 4), \dots, \Phi(6, 39), \Phi(11, 4), \dots, \Phi(36, 39).$
7. $\Phi(m_2, m_3)$, $8 \times 16 = 128$ elements; starting location: 488.

$\Phi(2, 3), \dots, \Phi(2, 38), \Phi(7, 3), \dots, \Phi(7, 38), \Phi(12, 3), \dots, \Phi(37, 38) \text{ .}$

$\Phi(2, 4), \dots, \Phi(2, 39), \Phi(7, 4), \dots, \Phi(7, 39), \Phi(12, 4), \dots, \Phi(37, 39) \text{ .}$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcImpulseResponse</i> or <i>pDstMatrix</i> pointer is NULL.

Codebook Gain Functions

These functions can be used to predict, quantize, decode, and control fixed- and adaptive-codebook gains both in G.729 encoding and decoding procedures.

DecodeGain_G729

Decodes the adaptive and fixed-codebook gains.

Syntax

```
IppStatus ippsDecodeGain_G729_16s (Ipp32s energy, Ipp16s* pPastEnergy, const
    Ipp16s* pQuaIndex, Ipp16s* pGain);
IppStatus ippsDecodeGain_G729I_16s(Ipp32s energy, Ipp16s valGainAttenuation,
    Ipp16s* pPastEnergy, const Ipp16s* pQuaIndex, Ipp16s* pGain);
```

Parameters

<i>energy</i>	Input energy of the codeword.
<i>pPastEnergy</i>	Pointer to the input/output vector (in Q14) of the log-energies for fixed codebook contributions of the 4 previous subframes.
<i>pQuaIndex</i>	NULL for frame erasure; otherwise, pointer to the vector of codebook indices: <i>pQuaIndex</i> [0] - first stage codebook index , <i>pQuaIndex</i> [1] - second stage codebook index .
<i>valGainAttenuation</i>	Attenuation factor for the gains in case of frame erasure (<i>pQuaIndex</i> = NULL).
<i>pGain</i>	Pointer to the input/output decoded gain: <i>pGain</i> [0] - adaptive (pitch) gain, <i>pGain</i> [1] - fixed codebook gain.

Description

The function `ippsDecodeGain_G729` is declared in the `ippsc.h` file. This function decodes the adaptive and fixed-codebook gains. The fixed codebook gain g_c can be expressed as follows:

$$g_c = \gamma \, g'_c ,$$

where g_c' is a predicted gain based on previous fixed codebook energies, and γ is a correction factor. The predicted gain g_c' is obtained by predicting the log-energy of the current fixed-codebook contribution from the log-energy of the previous fixed-codebook contribution, using the 4th order MA predictor with coefficients [0.68, 0.58, 0.34, 0.19].

The adaptive codebook gain and the factor are vector-quantized using conjugate structured codebooks. The first element in each codebook represents the quantized adaptive-codebook gain and the second element represents the quantized fixed-codebook. In case of frame erasure, the gains are the attenuated versions of the previous gains.

ippsDecodeGain_G729_16s. The two-stage conjugate structured two-dimensional codebook is used: first stage in 3 bits and second stage in 4 bits. In case of frame erasure, factors 0.9 and 0.98 in Q15 are used for attenuation of the adaptive and fixed codebook gains, respectively.

ippsDecodeGain_G729I_16s. The new 6-bit conjugate structured codebook is used. In case of frame erasure, the factor given by *valGainAttenuation* is used for attenuation of both the adaptive and fixed codebook gains.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pPastEnergy</i> or <i>pGain</i> pointer is NULL.

GainControl_G729

Calculates adaptive gain control.

Syntax

```
IppStatus ippsGainControl_G729_16s_I (const Ipp16s* pSrc, Ipp16s* pSrcDst,
    Ipp16s* pGain);
IppStatus ippsGainControl_G729A_16s_I (const Ipp16s* pSrc, Ipp16s* pSrcDst,
    Ipp16s* pGain);
IppStatus ippsGainControl_G729_32f_I (Ipp32f gainScalingFactor, Ipp32f
    gainFactor, Ipp32f* pSrcDst, Ipp32f* pGain);
```

Parameters

<i>pSrc</i>	Pointer to the source reconstructed speech vector [40].
-------------	---

<i>gainScalingFactor</i>	Gain scaling factor, which is the ratio of energies of the reconstructed speech signal and post-filtered output signal.
<i>gainFactor</i>	Gain adjustment factor.
<i>pSrcDst</i>	Pointer to the input post-filtered signal and output gain-compensated signal vector [40].
<i>pGain</i>	Pointer to the output adaptive gain.

Description

These functions are declared in the `ippsc.h` file. The function `ippsGainControl_G729` compensates the gain difference between the reconstructed speech signal *sr*, given by *pSrc* , and the filtered signal *sf*, given by *pSrcDst*. First, the gain factor *G* is calculated as follows:

$$G = \frac{\sum_{i=0}^{39} |sr[i]|}{\sum_{i=0}^{39} |sf[i]|}$$

The output gain *spf* for the post-filtered signal *pSrcDst* is calculated as:

$$spf(n) = g^{(n)} \cdot sf(n), \quad n = 0, \dots, 39,$$

where

$$g^{(n)} = 0.85 \cdot g^{(n-1)} + 0.15 \cdot G, \quad n = 0, \dots, 39, \quad g^{(-1)} = 1.0.$$

The function returns $g^{(39)}$ in *pGain*.

For the function `ippsGainControl_G729A_16s`, the gain factor *G* and factors $g^{(n)}$ are calculated differently as given by:

$$G = \sqrt{\frac{\sum_{i=0}^{39} |sr[i]|^2}{\sum_{i=0}^{39} |spf[i]|^2}}$$

$$g^{(n)} = 0.9 \cdot g^{(n-1)} + 0.1 \cdot G$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pSrcDst</code> or <code>pGain</code> pointer is NULL.

GainQuant_G729

Quantizes the codebook gain using a two-stage conjugate-structured codebook.

Syntax

```
IppStatus ippGainQuant_G729_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s*
    pSrcFltAdptVector, const Ipp16s* pSrcFixedVector, const Ipp16s*
    pSrcFltFixedVector, Ipp16s* pSrcDstEnergyErr, Ipp16s* pDstQGain, Ipp16s*
    pDstQGainIndex, Ipp16s tameProcess);

IppStatus ippGainQuant_G729D_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s*
    pSrcFltAdptVector, const Ipp16s* pSrcFixedVector, const Ipp16s*
    pSrcFltFixedVector, Ipp16s* pSrcDstEnergyErr, Ipp16s* pDstQGain, Ipp16s*
    pDstQGainIndex, Ipp16s tameProcess)
```

Parameters

<code>pSrcAdptTarget</code>	Pointer to the target signal for adaptive codebook search vector [40], in Q11.
<code>pSrcFltAdptVector</code>	Pointer to the filtered adaptive codebook vector [40], in Q11.
<code>pSrcFixedVector</code>	Pointer to the pre-weighted fixed-codebook vector [40], in Q12.
<code>pSrcFltFixedVector</code>	Pointer to the filtered fixed codebook vector [40], in Q12.
<code>pSrcDstEnergyErr</code>	Pointer to the previous predicted energy error vector [4], in Q10.
<code>pDstQGain</code>	Pointer to the quantized gain (G_p and G_c).
<code>pDstQGainIndex</code>	Pointer to the gain codebook indices, GA and GB .

tameProcess Taming process indicator.

Description

The functions `ippsGainQuant_G729` and `ippsGainQuant_G729D` are declared in the `ippsc.h` file. These functions use a two-stage conjugate-structured codebook. The function `ippsGainQuant_G729_16s` quantizes the gain using the 7-bit codebook, whereas the function `ippsGainQuant_G729D_16s` uses the 6-bit codebook. The gain codebook search is done by minimizing the mean-squared weighted error between original and reconstructed speech:

$$E = \mathbf{x}^t \mathbf{x} + g_p^2 \mathbf{y}^t \mathbf{y} + g_c^2 \mathbf{z}^t \mathbf{z} - 2g_p \mathbf{x}^t \mathbf{y} - 2g_c \mathbf{x}^t \mathbf{z} + 2g_p g_c \mathbf{y}^t \mathbf{z} ,$$

where \mathbf{x} is the adaptive target, \mathbf{y} is the filtered adaptive vector, and \mathbf{z} is the filtered fixed vector; g_p is the pitch gain, g_c is the fixed gain.

The functionality is as follows:

1. Calculate the average energy minus the mean energy of the fixed-codebook contribution:

$$E = 10 \log \left(\frac{1}{40} \sum_{n=0}^{39} c(n)^2 \right)$$

Then calculate $Eb - E = 30(dB) - E$.

2. Calculate the predicted energy from prediction error of the previous subframes:

$$\tilde{E}^m = \sum_{i=1}^4 b_i \hat{U}^{(m-i)}$$

\hat{U}^m is calculated by the equation: $U^m = E - \tilde{E}^m$.

3. Calculate g_c by the equation: $g_c = 10^{(\tilde{E}^m + Eb - E)/20}$.

4. Calculate the following elements:

$$G0 = \mathbf{y}^t \mathbf{y} , G1 = -2\mathbf{x}^t \mathbf{y} , G2 = \mathbf{z}^t \mathbf{z} , G3 = -2\mathbf{x}^t \mathbf{z} , G4 = 2\mathbf{y}^t \mathbf{z} .$$

5. Calculate optimum gain G_p and G_c as:

$$G_p = -\frac{2 \times G1 \times G2 - G3 \times G4}{4 \times G0 \times G2 - G4 \times G4} , G_c = -\frac{2 \times G0 \times G3 - G1 \times G4}{4 \times G0 \times G2 - G4 \times G4}$$

6. Pre-select a domain in which G_p and G_c are most possibly located. The output is *index0* and *index1*.

7. Select the gain vector in the codebook *GA* and *GB*, the range is limited by *index0* and *index1*. The search criteria is minimizing the following expression:

$$E = G0 \times G_p^2 + G2 \times G_c^2 + G1 \times G_p + G3 \times G_c + G4 \times G_p \times G_c .$$

8. Update U^m .

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcAdptTarget</i> , <i>pSrcFltAdptVector</i> , <i>pSrcFixedVector</i> , <i>pSrcFltFixedVector</i> , <i>pSrcDstEnergyErr</i> , <i>pDstQGain</i> or <i>pDstQGainIndex</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>tameProcess</i> is not 0 or 1.

AdaptiveCodebookContribution_G729

Updates target vector for codebook search by subtracting adaptive codebook contribution.

Syntax

```
IppStatus ippsAdaptiveCodebookContribution_G729_16s (Ipp16s gain, const Ipp16s*
    pFltAdptVector, const Ipp16s* pSrcAdptTarget, Ipp16s* pDstAdptTarget);
IppStatus ippsAdaptiveCodebookContribution_G729_32f (Ipp32f gain, const Ipp32f*
    pFltAdptVector, const Ipp32f* pSrcAdptTarget, Ipp32f* pDstAdptTarget);
```

Parameters

<i>gain</i>	Adaptive codebook gain g_p .
<i>pFltAdptVector</i>	Pointer to the input filtered adaptive codebook vector $y(n)$.
<i>pSrcAdptTarget</i>	Pointer to the input target vector $x(n)$.
<i>pDstAdptTarget</i>	Pointer to the output updated target vector $x'(n)$.

Description

The function `ippsAdaptiveCodebookContribution_G729` is declared in the `ippsc.h` file. This function subtracts the adaptive codebook contribution from the target signal and stores updated target vector as follows:

$$x'(n) = x(n) - g_p \cdot y(n), n = 0, \dots, 39$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pFltAdptVector</code> or <code>pSrcAdptTarget</code> or <code>pDstAdptTarget</code> pointer is NULL.

AdaptiveCodebookGain_G729

Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.

Syntax

```
IppStatus ippsAdaptiveCodebookGain_G729_16s(const Ipp16s* pSrcAdptTarget,
const Ipp16s* pSrcImpulseResponse, const Ipp16s* pSrcAdptVector, Ipp16s*
pDstFltAdptVector, Ipp16s* pResultAdptGain);

IppStatus ippsAdaptiveCodebookGain_G729A_16s(const Ipp16s* pSrcAdptTarget,
const Ipp16s* pSrcLPC, const Ipp16s* pSrcAdptVector, Ipp16s*
pDstFltAdptVector, Ipp16s* pResultAdptGain);
```

Parameters

<code>pSrcAdptTarget</code>	Pointer to the adaptive target signal vector [40].
<code>pSrcImpulseResponse</code>	Pointer to the impulse response of the perceptual weighting filter vector [40], in Q12.
<code>pSrcAdptVector</code>	Pointer to the adaptive-codebook vector [40], in Q12 (for <code>ippsAdaptiveCodebookGain_G729_16s</code>) or in Q10 (for <code>ippsAdaptiveCodebookGain_G729A_16s</code>).
<code>pSrcLPC</code>	Pointer to the LPC coefficients of the synthesis filter vector [11], in Q12.

pDstFltAdptVector Pointer to the filtered adaptive-codebook vector [40].

pResultAdptGain Pointer to the adaptive-codebook gain, in Q14.

Description

These functions are declared in the `ippsc.h` file. They compute the gain of the adaptive-codebook vector, and calculate the filtered adaptive-codebook vector, respectively. Both functions are applied in subframes.

The first function `ippsAdaptiveCodebookGain_G729` is designed for G.729/B codec and implements the following functionality:

1. First, convolve the adaptive-codebook vector, $v(n)$, with the impulse response, $h(n)$, to obtain the filtered adaptive-codebook vector, $y(n)$, using the formula:

$$y(n) = \sum_{i=0}^n v(i)h(n-i) , n = 0,1,...,39 .$$

2. Next, calculate the adaptive-codebook gain g_p as given by:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)} ,$$

where $x(n)$ is the adaptive target signal.

3. Finally, bound the adaptive-codebook gain g_p to the range $[0, 1.2]$.

The second function `ippsAdaptiveCodebookGain_G729A` is designed for G.729A codec and implements the following functionality:

1. First, pass the adaptive-codebook vector, $v(n)$, through the synthesis filter, $1/A_q(z/\gamma)$, to obtain the filtered adaptive-codebook vector, $y(n)$, using the formula :

$$y(n) = v(n) - \sum_{i=1}^{10} a_i y(n-i) , n = 0,1,...,39 .$$

2. Next, calculate the adaptive-codebook gain g_p as:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)},$$

where $x(n)$ is the adaptive target signal.

3. Finally, bound the adaptive-codebook gain g_p to the range $[0, 1.2]$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcAdptTarget</i> , <i>pSrcImpulseResponse</i> , <i>pSrcAdptVector</i> , <i>pSrcLPC</i> , <i>pDstFltAdptVector</i> , or <i>pResultAdptGain</i> pointer is NULL.

Filter Functions

These functions are used to perform different types of filtering, including high pass filtering on pre-processing stage of encoding and post-processing stage of decoding, as well as the long-term and short-term postfilter in decoding.

The residual (FIR), simple synthesis (IIR), harmonic, and preemphasize filter functions may be combined to perform more complex synthesis filtering and may be used in different encode and decode stages.

ResidualFilter_G729

Implements an inverse LP filter and obtains the residual signal.

Syntax

```
IppStatus ippsResidualFilter_G729_16s(const Ipp16s* pSrcSpch, const Ipp16s*
    pSrcLPC, Ipp16s* pDstResidual);
IppStatus ippsResidualFilter_G729E_16s(const Ipp16s* pCoeffs, int order, const
    Ipp16s* pSrc, Ipp16s* pDst, int len)
```

Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal. Elements <i>pSrcSpch</i> [0...39] are the present speech signals, whereas <i>pSrcSpch</i> [-10... -1] are the history to be used.
<i>pSrcLPC</i>	Pointer to the LP coefficients vector [11], in Q12.
<i>pDstResidual</i>	Pointer to the LP residual.
<i>pCoeffs</i>	Pointer to the vector [<i>order</i> +1] of filter coefficients.
<i>order</i>	The filter order.
<i>pSrc</i>	Pointer to the source vector [<i>len</i>]. Values <i>pSrc</i> [- <i>order</i> ,...,-1] must also be supplied.
<i>pDst</i>	Pointer to the output filtered residual vector [<i>len</i>].
<i>len</i>	Length of the source and destination vectors.

Description

The functions `ippsResidualFilter_G729` and `ippsResidualFilter_G729E` are declared in the `ippsc.h` file.

ippsResidualFilter_G729. This function filters the input speech signal through an inverse LP filter and gets the residual signal as:

$$r(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i \times s(n-i), n = 0,1,...,39.$$

ippResidualFilter_G729E. This function implements the same operation as the previous function, but accepts the variable filter order.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcSpch</i> , <i>pSrcLPC</i> , <i>pDstResidual</i> , <i>pSrc</i> , <i>pCoeffs</i> , or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>order</i> is less than or equal to 0, or when <i>len</i> is less than or equal to 0 and <code>valLPCOrder > len</code> .

SynthesisFilter_G729

*Reconstructs the speech signal
from LP coefficients and residuals.*

Syntax

```
IppStatus ippsSynthesisFilter_G729_16s(const Ipp16s* pSrcResidual, const
    Ipp16s* pSrcLPC, Ipp16s* pSrcDstSpch);
IppStatus ippsSynthesisFilterZeroStateResponse_NR_16s(const Ipp16s* pSrcLPC,
    Ipp16s* pDstImp, int len, int scaleFactor);
IppStatus ippsSynthesisFilter_G729E_16s(const Ipp16s* pLPC, int order, const
    Ipp16s* pSrc, Ipp16s* pDst, int len, const Ipp16s* pMem);
IppStatus ippsSynthesisFilter_G729E_16s_I(const Ipp16s* pLPC, int order,
    Ipp16s* pSrcDst, int len, const Ipp16s* pMem);
IppStatus ippsSynthesisFilter_G729_32f(const Ipp32f* pLpc, int order, const
    Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f* pMem);
```

Parameters

<i>pSrcResidual</i>	Pointer to the LP residual signal vector [40].
<i>pSrcLPC</i>	Pointer to the LP coefficients vector [11], in Q12 for 16s data.
<i>pSrcDstSpch</i>	Pointer to the synthesized and updated speech. Elements <i>pSrcDstSpch</i> [0...39] are the present synthesized speech, <i>pSrcDstSpch</i> [-10...-1] are the history to be used.
<i>pLPC</i>	Pointer to the LP coefficients vector [<i>order</i> +1].

<i>order</i>	The filter order (<i>pLPC</i> [<i>order</i> +1] vector must be supplied on input) .
<i>pSrc</i>	Pointer to the source vector [<i>len</i>].
<i>pDst</i>	Pointer to the destination speech vector [<i>len</i>].
<i>pSrcDst</i>	Pointer to the source and destination vector [<i>len</i>].
<i>pDstImp</i>	Pointer to the destination zero state impulse response vectore [<i>len</i>]
<i>len</i>	Length of the source and destination vectors.
<i>pMem</i>	Pointer to the filter memory vector [<i>order</i>] supplied for filtering.
<i>scaleFactor</i>	Scale factor for the input LP coefficients and output impulse response, accordingly.

Description

These functions are declared in the `ippsc.h` file.

The function `ippsSynthesisFilter_G729_16s` uses the default LP filter of 10-th order to reconstruct speech signal from the residual signal as:

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i) , n = 0,1,...,39 .$$

The function `ippsSynthesisFilterZeroStateResponse_NR_16s` performs the same operation as `ippsSynthesisFilter_NR_16s` for the same parameters, but uses no memory and processes the impulse input vector *pSrc* which is set as

$$\begin{aligned} pSrc[0] &= 2^{scaleFactor}, \\ pSrc[i] &= 0 , i = 1,... len - 1 . \end{aligned}$$

The functions `ippsSynthesisFilter_G729E_16s`, `ippsSynthesisFilter_G729E_16s_I`, and `ippsSynthesisFilter_G729_32f` perform synthesis filtering of the order given by the parameter *order* and operate like `ippsSynthesisFilter_G729_16s` function for the default order 10.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the pointers is NULL.
<code>ippStsOverflow</code>	Indicates a warning that an overflow has occurred.

`ippStsSizeErr` Indicates an error when *order* is less than or equal to 0.

LongTermPostFilter_G729

*Restores the long-term information
from the old speech signal.*

Syntax

```
IppStatus ippsLongTermPostFilter_G729_16s (Ipp16s gammaFactor, int valDelay,
      const Ipp16s* pSrcDstResidual, Ipp16s* pDstFltResidual,
      Ipp16s* pResultVoice);

IppStatus ippsLongTermPostFilter_G729A_16s(Ipp16s valDelay, const Ipp16s*
      pSrcSpch, const Ipp16s* pSrcLPC, Ipp16s* pSrcDstResidual, Ipp16s*
      pDstFltResidual);

IppStatus ippsLongTermPostFilter_G729B_16s(Ipp16s valDelay, const Ipp16s*
      pSrcSpch, const Ipp16s* pSrcLPC, Ipp16s* pSrcDstResidual, Ipp16s*
      pDstFltResidual, Ipp16s* pResultVoice, Ipp16s frameType);
```

Parameters

<i>gammaFactor</i>	Post filter coefficient y_p in Q15.
<i>valDelay</i>	Pitch delay.
<i>pSrcSpch</i>	Pointer to the reconstructed speech $\hat{s}(n)$, in Q15. Elements <i>pSrcSpch</i> [0...39] are the present speech signals, elements <i>pSrcSpch</i> [-10...-1] are the history that will be used.
<i>pSrcLPC</i>	Pointer to the LP coefficients \hat{a}_i vector [11], in Q12.
<i>pSrcDstResidual</i>	Pointer to the residual after long term prediction vector [40], elements [-152...-1] are the long term prediction history that will be used.
<i>pDstFltResidual</i>	Pointer to the output filtered residual vector [40].
<i>pResultVoice</i>	Pointer to the voice information.
<i>frameType</i>	The type of the frame: 0 - stands for untransmitted frame, 1 - stands for normal speech frame, 2 - stands for SID frame.

Description

These functions are declared in the `ippssc.h` file. These FIR filters are used to restore the long-term relationship from old speech at the length of pitch. They are applied in subframes.

The first function **ippsLongTermPostFilter_G729** has the following functionality:

1. First, calculate autocorrelation $R(k)$ of $\hat{x}(n)$ and find the best integer $T0$, in the range of $[\text{int}(T1)-1, \text{int}(T1)+1]$, which maximizes $R(k)$ using the following formula:

$$R(k) = \sum_{n=0}^{39} \hat{x}(n) \cdot \hat{x}(n-k) \quad (9-6)$$

Here $\text{int}(T1)$ is the integer part of the pitch delay $T1$ in the first subframe.

2. Second, seek for the best fractional delay T with resolution $1/8$ around $T0$. This is done by finding the delay with the highest pseudo-normalized correlation $R'(k)$:

$$R'(k) = \frac{\sum_{n=0}^{39} \hat{x}(n) \cdot \hat{x}_k(n)}{\sqrt{\sum_{n=0}^{39} \hat{x}_k(n) \cdot \hat{x}_k(n)}} ,$$

where $\hat{x}_k(n)$ is the residual at delay k .

3. Once the optimal delay T is found, determine whether the long-term filter should be disabled or not, using the following condition:

$$\text{if } \frac{R'(T)^2}{\sum_{n=0}^{39} \hat{x}(n) \cdot \hat{x}(n)} < 0.5 , \text{ let } g_1 = 0 ; \quad (9-7)$$

$$\text{otherwise, let } g_1 = \frac{\sum_{n=0}^{39} \hat{x}(n) \cdot \hat{x}_k(n)}{\sum_{n=0}^{39} \hat{x}_k(n) \cdot \hat{x}_k(n)} , \text{ and bound it to } [0, 1.0] \quad (9-8)$$

4. Finally, use the following FIR filter:

$$y(n) = \frac{1}{1 + \gamma_p \cdot g_1} (\hat{x}(n) + \gamma_p \cdot g_1 \cdot \hat{x}(n-T)) , \quad (9-9)$$

where $\hat{x}(n)$ is the residual signal and the factor γ_p controls the amount of long-term post-filtering.

The second function **ippsLongTermPostFilter_G729A** is designed for G.729A codec, and performs the following operations:

1. First, get the residual $\hat{x}(n)$ from synthesized speech $\hat{s}(n)$ using the formula:

$$\hat{x}(n) = \hat{s}(n) + \sum_{i=1}^{10} \gamma_n^i \cdot \hat{a}_i \cdot \hat{s}(n-i) , \quad (9-5)$$

where $\hat{s}(n)$ is the synthesized speech, \hat{a}_i are the quantized LP coefficients and

$$\gamma_n = 0.55.$$

2. Second, calculate autocorrelation $R(k)$ of $\hat{x}(n)$ and find the best integer T in the range $[\text{int}(T1)-3, \text{int}(T1)+3]$, which maximizes $R(k)$ using the formula (9-6).

Here $\text{int}(T1)$ is the integer part of the pitch delay $T1$ in the first subframe.

3. Once the optimal delay T is found, determine whether the long-term filter should be disabled or not, using the relations (9-7) and (9-8).

6. Finally, use the FIR filter (9-9).

The third function **ippsLongTermPostFilter_G729B** is actually the combination of other functions as demonstrated by the code below:

```
{
const short g_pst[11] = {32767,18022,9912,5451,2998,1649,907,499,274,151,83};
/* y_n = 0.55 powered by i=0,1,...,10 in Q15 */
short coeffs[LP_ORDER+1];          /* temporary nominator coefficients */
short vc;

ippsMul_NR_16s_Sfs(g_pst, pSrcLpc, coeffs, LP_11, 15);
ippsResidualFilter_G729_16s(pSrcSpch, coeffs, pSrcDstResidual+154);
if (1 == frameType){
```

```

        ippsLongTermPostFilter_G729_16s (16384, valDelay, pSrcDstResidual + 154,
            pDstFltResidual, &vc); /* Harmonic filtering : gp=0.5 in Q15*/
        *pResultVoice = (vc != 0);
    }else{
        ippsCopy_16s(pSrcDstResidual + 154, pDstFltResidual, 40);
        *pResultVoice = 0;
    }
}

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcSpch</i> , <i>pSrcLPC</i> , <i>pSrcDstResidual</i> , <i>pDstFltResidual</i> or <i>pResultVoice</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>valDelay</i> is not in the range [18, 145] for <code>ippsLongTermPostFilter_G729A</code> function, or <i>valDelay</i> is not in the range [0, 143] for <code>ippsLongTermPostFilter_G729B</code> function, or <i>frameType</i> is not in the range [0, 2].

ShortTermPostFilter_G729

Restores speech signal from the residuals.

Syntax

```

IppStatus ippsShortTermPostFilter_G729_16s(const Ipp16s* pSrcLPC, const
    Ipp16s* pSrcFltResidual, Ipp16s* pSrcDstSpch, Ipp16s*
    pDstImpulseResponse);

IppStatus ippsShortTermPostFilter_G729A_16s(const Ipp16s* pSrcLPC, const
    Ipp16s* pSrcFltResidual, Ipp16s* pSrcDstSpch);

```

Parameters

<i>pSrcLPC</i>	Pointer to the quantized LP coefficients vector [11], in Q12.
<i>pSrcFltResidual</i>	Pointer to the residual signal $x(n)$ vector [40], in Q15.

pSrcDstSpch Pointer to the short-term filtered speech $y(n)$, in Q15. Elements *pSrcDstSpch*[0...39] are the present short-term filtered speech signals, elements *pSrcDstSpch*[-10...-1] are the history that will be used.

pDstImpulseResponse Pointer to the generated impulse response $h_f(n)$ vector [20], in Q12.

Description

These functions are declared in the `ippsc.h` file. These two IIR filters restore the speech from the residual. They both are applied in subframes.

The first function `ippsShortTermPostFilter_G729` is designed for G.729/B codec, and has the following functionality:

1. First, compute the impulse response $h_f(n)$ of the inverse filter. $\hat{A}(z)$.
2. Next, calculate the gain term g_f using the formula:

$$g_f = \sum_{n=0}^{10} |h_f(n)|$$

3. Finally, filter the residual through the inverse filter $\hat{A}(z)$, as follows:

$$y(n) = \frac{1}{g_f} x(n) - \sum_{i=1}^{10} \hat{a}_i y(n-i), \quad n = 0, 1, \dots, 39,$$

where $x(n)$ is the residual and \hat{a}_i are the weighted quantized LP coefficients.

The second function `ippsShortTermPostFilter_G729A` is designed for G.729A codec. This function filters the residual through the inverse filter $\hat{A}(z)$, using the formula:

$$y(n) = x(n) - \sum_{i=1}^{10} \hat{a}_i y(n-i), \quad n = 0, 1, \dots, 39,$$

where $x(n)$ is the residual and \hat{a}_i are the weighted quantized LP coefficients.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcLPC</code> , <code>pSrcFltResidual</code> , <code>pSrcDstSpch</code> , or <code>pDstImpulseResponse</code> pointer is NULL.

TiltCompensation_G729

Compensates for the tilt in the short-term filter.

Syntax

```
IppStatus ippstTiltCompensation_G729_16s(const Ipp16s* pSrcImpulseResponse,
    Ipp16s* pSrcDstSpch);

IppStatus ippstTiltCompensation_G729A_16s(const Ipp16s* pSrcLPC, Ipp16s*
    pSrcDstFltResidual);

IppStatus ippstTiltCompensation_G729E_16s(Ipp16s val, const Ipp16s* pSrc,
    Ipp16s* pDst);
```

Parameters

<code>pSrcImpulseResponse</code>	Pointer to the impulse response $h_f(n)$ vector [20], in Q12.
<code>pSrcLPC</code>	Pointer to the gamma weighted LP coefficients vector [22], in Q12. The first 11 elements refer to γ_n and the next 11 elements refer to γ_d .
<code>pSrcDstSpch</code>	Pointer to the present and tilt-compensated speech $x(n)$, in Q15. Elements <code>pSrcDstSpch[0...39]</code> are the present speech signals, element <code>pSrcDstSpch[-1]</code> is the history that will be used.
<code>pSrcDstFltResidual</code>	Pointer to the long-term filtered LP residual, in Q15. Elements <code>pSrcDstFltResidual[0...39]</code> are the present long-term filtered LP residual signals, element <code>pSrcDstFltResidual[-1]</code> is the history that will be used.
<code>val</code>	The input tilt factor.
<code>pSrc</code>	Pointer to the source vector [41].
<code>pDst</code>	Pointer to the output filtered residual vector [40]

Description

These functions are declared in the `ippsc.h` file. These FIR filters compensate for the tilt in the short-term filter. They are applied in subframes.

The function `ippSTiltCompensation_G729` is designed for G.729/B codec, and the functionality is as follows.

1. First, calculate correlation $r_h(i)$ of the impulse response $h_f(i)$ as:

$$r_h(i) = \sum_{j=0}^{19-i} h_f(j)h_f(j+i)$$

Then obtain the tilt factor k_1' using the relation:

$$k_1' = -\frac{r_h(1)}{r_h(0)}$$

2. Next, obtain the gain term g_t using the formula:

$$g_t = 1 - |\gamma_t \cdot k_1'| ,$$

where $\gamma_t = 0.9$ if k_1' is negative, and $\gamma_t = 0.2$ if k_1' is positive.

3. Finally, filter the speech using the formula:

$$y(n) = \frac{1}{g_t}(x(n) + \gamma_t \cdot k_1' \cdot x(n-1)) ,$$

where $x(n)$ is the input speech signal.

The function `ippSTiltCompensation_G729E` performs only stages 2 and 3 for a given input tilt factor `val` and filter memory specified by `pSrc[0]`.

The function `ippSTiltCompensation_G729A` is designed for G.729A codec, and the functionality is as follows.

1. First, compute the impulse response $h_f(n)$ of the inverse filter. $\hat{A}(z)$.
2. Next, calculate correlation $r_h(i)$ of the impulse response $h_f(i)$ as:

$$r_h(i) = \sum_{j=0}^{21-i} h_f(j)h_f(j+i)$$

Then obtain the tilt factor k_1' , using the relation:

$$k_1' = -\frac{r_h(1)}{r_h(0)}$$

3. Finally, filter the residual using the formula:

$$y(n) = x(n) + \gamma_t \cdot k_1' \cdot x(n-1),$$

where $x(n)$ is the residual, and $\gamma_t = 0.8$ if k_1' is negative; or $\gamma_t = 0$ if k_1' is positive.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDstSpch</i> , <i>pSrcImpulseResponse</i> , <i>pSrcLPC</i> , <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDstFltResidual</i> pointer is NULL.

HarmonicFilter

Calculates the harmonic filter.

Syntax

```
IppStatus ippsharmonicFilter_16s_I (Ipp16s beta, int T, Ipp16s* pSrcDst,
    int len);
IppStatus ippsharmonicFilter_NR_16s (Ipp16s beta, int T, const Ipp16s* pSrc,
    Ipp16s* pDst, int len);
IppStatus ippsharmonicFilter_32f_I(Ipp32f beta, int T, Ipp32f* pSrcDst, int
    len);
```

Parameters

<i>beta</i>	The beta factor, in Q15 for 16s data.
<i>T</i>	The integer delay.
<i>pSrc</i>	Pointer to the source vector. Elements <i>pSrc</i> [- <i>T</i> ,...,0,..., <i>len</i> -1] will be used as input.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector. Elements <i>pSrcDst</i> [- <i>T</i> ,...,0,..., <i>len</i> -1] will be used as input, elements <i>pSrcDst</i> [0,..., <i>len</i> -1] will be computed.

len Number of elements in the source and destination vectors.

Description

These functions are declared in the `ippsc.h` file.

The functions `ippsHarmonicFilter_16s_I` and `ippsHarmonicFilter_32f_I` implement the adaptive pre-filter:

$$H(z) = \frac{1}{1 - \beta \cdot z^{-T}},$$

which can be used to enhance the harmonic component of speech.

The calculation is as follows:

$$pSrcDst[n] = pSrcDst[n] + \beta \cdot pSrcDst[n-T], \quad 0 \leq n < len.$$

The function `ippsHarmonicFilter_NR_16s` implements the harmonic noise shaping filter:

$$H(z) = 1 + \beta \cdot z^{-T}$$

The calculation is as follows:

$$pDst[n] = pSrc[n] + \beta \cdot pSrc[n-T], \quad 0 \leq n < len.$$

Both filters can be used to enhance the harmonic component of the input signal.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

HighPassFilterSize_G729

Returns the G729 high-pass filter size.

Syntax

```
IppStatus ippsHighPassFilterSize_G729 (int* pSize);
```

Parameters

pSize Pointer to the output value of the memory size.

Discussion

The function `ippsHighPassFilterSize_G729` is declared in the `ippsc.h` file. This function returns the size of memory required for the `ippsHighPassFilterInit_G729` function to operate.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pSize* pointer is NULL.

HighPassFilterInit_G729

Initializes the high-pass filter.

Syntax

```
IppStatus ippsHighPassFilterInit_G729 (Ipp16s* pCoeff, char* pMemUpdated);
```

Parameters

pCoeff Pointer to the array of length 6 containing values $a_0, a_1, a_2, b_0, b_1, b_2$ in Q12 or Q13.

pMemUpdated Pointer to the memory allocated for the filter.

Description

The function `ippsHighPassFilterInit_G729` is declared in the `ippsc.h` file. This function initializes the internal data of the high-pass filter: a_0 should be equal to scaled 1 (in Q12 or Q13). The filter history data $x_{-2}, x_{-1}, y_{-2}, y_{-1}$ are set to zero. The `ippsHighPassFilter_G729_16s_ISfs` function uses this memory for filtering.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pCoeff* or *pMemUpdated* pointer is NULL.

HighPassFilter_G729

Performs G729 high-pass filtering.

Syntax

```
IppStatus ippsHighPassFilter_G729_16s_ISfs (Ipp16s* pSrcDst, int len, int
    scaleFactor, char* pMemUpdated);
```

Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector [<i>len</i>].
<i>len</i>	Number of elements in the source and destination vectors.
<i>scaleFactor</i>	Scale factor for output data scaling.
<i>pMemUpdated</i>	Pointer to the memory allocated for the filter.

Description

The function `ippsHighPassFilter_G729` is declared in the `ippsc.h` file. This function performs the input signal pre-processing or the output signal post-processing using the high-pass filter:

$$H_h(z) = \frac{b_0 + b_1 \cdot z^{-1} + b_2 \cdot z^{-2}}{a_0 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2}}$$

Currently, only $a_0 = 1$ (in Q12 or Q13) is supported. The function uses the scale factor value equal to 12 for input data pre-filtering, and equal to 13 for output data post-filtering.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> or <i>pMemUpdated</i> pointer is NULL.
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is not equal to 12 or 13.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

IIR16s_G729

Performs IIR filtering.

Syntax

```
IppStatus ippsIIR16sLow_G729_16s(const Ipp16s* pCoeffs, const Ipp16s* pSrc,
    Ipp16s* pDst, Ipp16s* pMem);
IppStatus ippsIIR16s_G729_16s(const Ipp16s* pCoeffs, const Ipp16s* pSrc,
    Ipp16s* pDst, Ipp16s* pMem);
```

Parameters

<i>pCoeffs</i>	Pointer to the input vector of filter coefficients vector [20]: $b_1, \dots, b_{10}, a_1, \dots, a_{10}$.
<i>pSrc</i>	Pointer to the input signal vector [40].
<i>pDst</i>	Pointer to the output filtered vector [40].
<i>pMem</i>	Pointer to the filter memory vector [20]. This vector must be initially filled with zeroes.

Description

The functions `ippsIIR16sLow_G729s_16s` and `ippsIIR16s_G729_16s` are declared in the `ippsc.h` file.

These functions perform infinite impulse response (IIR) filtering using the following transfer function:

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The function `ippsIIR16sLow_G729_16s` does not check for overflow and result saturation, whereas the function `ippsIIR16s_G729_16s` makes these checks.

The filter memory is updated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pCoeffs</i> , <i>pSrc</i> , <i>pDst</i> , or <i>pMem</i> pointer is NULL.

PhaseDispersionGetStateSize_G729D

Queries the memory length of the phase dispersion filter.

Syntax

```
IppStatus ippPhaseDispersionGetStateSize_G729D_16s (int* pSize);
```

Parameters

pSize Pointer to the output phase dispersion filter memory length in bytes.

Description

The function `ippPhaseDispersionGetStateSize_G729D_16s` reports the size of memory that the phase dispersion filter needs for proper operation.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.

PhaseDispersionInit_G729D

Initializes the phase dispersion filter memory.

Syntax

```
IppStatus ippPhaseDispersionInit_G729D_16s  
    (IppsPhaseDispersion16s_State_G729D* pPhDMem);
```

Parameters

pPhDMem Pointer to the memory of the phase dispersion filter.

Description

The function `ippsPhaseDispersionInit_G729D_16s` sets the contents of the given buffer as memory of the phase dispersion filter in initial state.

Return Values

`ippStsNoErr` Indicates no error.
`ippStsNullPtrErr` Indicates an error when the *pPhDMem* pointer is NULL.

PhaseDispersionUpdate_G729D

Updates the phase dispersion filter state.

Syntax

```
IppStatus ippsPhaseDispersionUpdate_G729D_16s(Ipp16s valPitchGain,  
Ipp16s valCodebookGain, IppsPhaseDispersion16s_State_G729D* pPhDMem);
```

Parameters

valPitchGain Long term pitch gain.
valCodebookGain Codebook gain
pPhDMem Pointer to the memory of phase dispersion filter.

Description

The function `ippsPhaseDispersionUpdate_G729D_16s` updates the state of phase dispersion filter memory with given pitch and codebook gains.

Return Values

`ippStsNoErr` Indicates no error.
`ippStsNullPtrErr` Indicates an error when the *pPhDMem* pointer is NULL.

PhaseDispersion_G729D

Performs the phase dispersion filtering.

Syntax

```
IppStatus ippsPhaseDispersion_G729D_16s(Ipp16s valCodebookGain, Ipp16s  
    valPitchGain, const Ipp16s* pSrcExcSignal, Ipp16s* pDstFltExcSignal,  
    Ipp16s* pSrcDstInnovation, IppsPhaseDispersion16s_State_G729D* pPhDMem)
```

Parameters

<i>valPitchGain</i>	Input long term pitch gain.
<i>valCodebookGain</i>	Input codebook gain
<i>pSrcExcSignal</i>	Pointer to the input excitation vector [40].
<i>pDstFltExcSignal</i>	Pointer to the output filtered excitation vector [40].
<i>pSrcDstInnovation</i>	Pointer to the input/output innovative codebook vector [40].
<i>pPhDMem</i>	Pointer to the memory of phase dispersion filter.

Description

The function `ippsPhaseDispersion_G729D_16s` performs the phase dispersion filtering for the given pitch and codebook gains. The filter alters the innovation signal (mainly its phase), such that a new innovation signal is created with the energy being more spread over the subframe. The filtering is performed by a circular convolution using one of the three "semi-random" impulse responses according to the different amounts of spreading.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcExcSignal</i> , <i>pDstFltExcSignal</i> , <i>pSrcDstInnovation</i> or <i>pPhDMem</i> pointer is NULL.

Preemphasize_G729A

Computes pre-emphasis of a post filter.

Syntax

```
IppStatus ippsPreemphasize_G729A_16s (Ipp16s gamma, const Ipp16s* pSrc, Ipp16s*
    pDst, int len, Ipp16s* pMem);

IppStatus ippsPreemphasize_G729A_16s_I (Ipp16s gamma, Ipp16s* pSrcDst, int len,
    Ipp16s* pMem);

IppStatus ippsPreemphasize_32f_I(Ipp32f gamma, Ipp32f* pSrcDst, int len,
    Ipp32f* pMem);
```

Parameters

<i>gamma</i>	The filter coefficient, in Q15 for 16s data.
<i>pSrc</i>	Pointer to the source vector, in Q0.
<i>pDst</i>	Pointer to the destination vector, in Q0.
<i>pSrcDst</i>	Pointer to the source and destination vector, in Q0.
<i>len</i>	Number of elements in the source and destination vectors.
<i>pMem</i>	Pointer to the filter memory [1].

Description

These functions are declared in the `ippsc.h` file and compute pre-emphasis of a post filter.

The computation is performed according to the difference signal pre-emphasis equation:

$$H(z) = 1 - \text{gamma} \cdot z^{-1}$$

The filter memory *pMem* should be initialized to zero and then be constantly updated while filtering.

The functions `ippsPreemphasize_G729A` perform no rounding (clipping).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , <i>pSrcDst</i> , or <i>pMem</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

WinHybridGetStateSize_G729E

Queries the length of the hybrid windowing module memory.

Syntax

```
IppStatus ippWinHybridGetStateSize_G729E_16s (int* pSize);
IppStatus ippWinHybridGetStateSize_G729E_32f (int* pSize);
```

Parameters

<i>pSize</i>	Pointer to the output length of hybrid windowing module memory in bytes.
--------------	--

Description

The function `ippWinHybridGetStateSize_G729E` reports the size of memory that the hybrid windowing module needs for proper operation.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.

WinHybridInit_G729E

Initializes the hybrid windowing module memory.

Syntax

```
IppStatus ippWinHybridInit_G729E_16s (IppsWinHybridState_G729E_16s* pMem);
IppStatus ippWinHybridInit_G729E_32f (IppsWinHybridState_G729E_32f* pMem);
```

Parameters

pMem Pointer to the memory of hybrid windowing module.

Description

The function `ippsWinHybridInit_G729E` sets the contents of the given buffer as the memory of the hybrid windowing module in initial state.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pMem* pointer is NULL.

WinHybrid_G729E

Applies the hybrid window and computes autocorrelation coefficients.

Syntax

```
IppStatus ippsWinHybrid_G729E_16s32s (const Ipp16s* pSrcSynthSpeech, Ipp32s*
    pDstInvAutoCorr, IppsWinHybridState_G729E_16s* pMem);

IppStatus ippsWinHybrid_G729E_32f (const Ipp32f* pSrcSynthSpeech, Ipp32f*
    pDstInvAutoCorr, IppsWinHybridState_G729E_32f* pMem);
```

Parameters

pSrcSynthSpeech Pointer to the input synthesized speech vector [144].

pDstInvAutoCorr Pointer to the output autocorrelation vector [31].

pMem Pointer to the memory of hybrid windowing module.

Description

The function `ippsWinHybrid_G729E` applies the hybrid window to input synthesized speech and computes autocorrelation coefficients as follows.

1. First, the input speech vector is multiplied by the hybrid window:

$$s_{w+L}(k) = s_w(k) \cdot w(144 - k), k = 0, \dots, 144$$

where

$$w(k) = \begin{cases} \sin((k+1) * 0.047783), k = 0, \dots, 34 \\ \sin(36 * 0.047783) * a^{(35-k)}, k = 35, \dots, 144 \end{cases}$$

Here $a = 0.9928337491$ so that $a^{40} = 0.75$ and $a^{160} = 0.75^4 = 0.31640625$.

2. Second, the autocorrelation of the windowed speech is calculated by the formulae:

$$R_{rec}(n) = \sum_{k=0}^{k=79} s(k+30) \cdot s(k+30-n), n = 0, \dots, 30$$

$$R(n) = \sum_{k=0}^{k=34} s(k+110) \cdot s(k+110-n), n = 0, \dots, 30$$

3. Third, the output autocorrelation is computed and the memory is updated as follows:

$$\begin{cases} mem(k) = 0.31640625 * mem(k) + R_{rec}(k) \\ pDstAutoCorr(k) = mem(k) + R(k) \end{cases}, k = 0, \dots, 30$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcSynthSpeech</code> , <code>pDstInvAutoCorr</code> or <code>pMem</code> pointer is NULL.

RandomNoiseExcitation_G729B

Initializes a random vector with a Gaussian distribution.

Syntax

```
IppStatus ippRandomNoiseExcitation_G729B_16s (Ipp16s* pSeed, Ipp16s* pExc, int len);
```

```
IppStatus ippsRandomNoiseExcitation_G729B_16s32f(Ipp16s* pSeed, Ipp32f* pExc,
int len);
```

Parameters

<i>pSeed</i>	Pointer to the input/output seed of random generator.
<i>pExc</i>	Pointer to the destination vector.
<i>len</i>	Length of the destination vector.

Description

The function `ippsRandomNoiseExcitation_G729B_16s` is declared in the `ippsc.h` file. This function generates random excitations as follows:

$$pExc[n] = \frac{\sum_{i=1}^{12} \xi_{(12n+i)}}{128}, n = 0, \dots, len-1$$

where $\xi_{k+1} = 31821 \cdot \xi_k + 13849$, $\xi_0 = *pSeed$

The seed of the random generator ξ_k is updated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pExc</i> or <i>pSeed</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

FilteredExcitation_G729

Computes filtered excitation.

Syntax

```
IppStatus ippsFilteredExcitation_G729_32f(const Ipp32f* pSrcImpulseResponse,
Ipp32f* pSrcDstPastExc, Ipp32f valExc, Ipp32s len);
```

Parameters

pSrcImpulseResponse Impulse response [*len*].

<i>pSrcDstPastExc</i>	Pointer to the previous and updated excitation vector [<i>len</i>].
<i>valExc</i>	Excitation.
<i>len</i>	Length

Description

The function computes filtered excitation using following formula:

$$\begin{aligned} pSrcDstPastExc[i] &= SrcDstPastExc[i-1] + valExc * pSrcImpulseResponse[i], \\ pSrcDstPastExc[0] &= valExc, \\ \text{where } i &= len - 1, \dots, 1. \end{aligned}$$

Return Value

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when the <i>pSrcImpulseResponse</i> or <i>pSrcDstPastExc</i> pointer is NULL.
<i>ippStsSizeErr</i>	Indicates an error when <i>len</i> is less than or equal to zero.

G.723.1 Related Functions

Intel IPP functions described in this section implement building blocks that can be used to create speech codecs compliant with the ITU-T Recommendation G.723.1 (see [ITU723](#), [ITU723A](#)). The list of these functions is given in the following table:

Table 9-3 Intel IPP G.723.1 Related Functions

Function Base Name	Operation
Linear Prediction Analysis Functions	
AutoCorr_G723	Estimates the auto-correlation of a vector.
AutoCorr_NormE_G723	Estimates normal auto-correlation of a vector.
LevinsonDurbin_G723	Calculates the LP coefficients from the autocorrelation coefficients.
LPCToLSF_G723	Converts LP coefficients to LSF coefficients.
LSFToLPC_G723	Converts LSF coefficients to the LP coefficients.
LSFDecode_G723	Performs inverse quantization of LSFs.
LSFQuant_G723	Quantizes LSF coefficients.

Table 9-3 Intel IPP G.723.1 Related Functions (continued)

Function Base Name	Operation
Codebook Search Functions	
OpenLoopPitchSearch_G723	Searches for an optimal pitch value.
ACELPFixedCodebookSearch_G723	Searches the ACELP fixed codebook for the excitation.
AdaptiveCodebookSearch_G723	Searches for the close loop pitch and the adaptive gain index.
MPMLQFixedCodebookSearch_G723	Searches the MP-MLQ fixed codebook for the excitation.
ToeplitzMatrix_G723	Calculates 416 elements of the Toeplitz matrix for fixed codebook search.
Gain Quantization Functions	
GainQuant_G723	Implements MP-MLQ gain estimation and quantization.
GainControl_G723	Extracts delayed pitch contribution.
Filter Functions	
HighPassFilter_G723	Performs high-pass filtering of the input signal.
IIR16s_G723	Performs IIR filtering.
SynthesisFilter_G723	Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).
TiltCompensation_G723	Computes tilt compensation filter.
HarmonicSearch_G723	Searches for the harmonic delay and gain for the harmonic noise shaping filter.
HarmonicNoiseSubtract_G723	Performs harmonic noise shaping.
DecodeAdaptiveVector_G723	Restores the adaptive codebook vector from excitation, pitch, and adaptive gain.
PitchPostFilter_G723	Calculates coefficients of the pitch post filter.

These functions are used in the Intel IPP *G.723.1 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Linear Prediction Analysis Functions

These functions perform LPC analysis, LSP coding (quantization) and decoding, as well as transformation between LPC and LSF coefficients.

AutoCorr_G723

Estimates the auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorr_G723_16s(const Ipp16s* pSrcSpch, Ipp16s*
    pResultAutoCorrExp, Ipp16s* pDstAutoCorr);
```

Parameters

pSrcSpch Pointer to the input speech signal vector [180].

pResultAutoCorrExp Pointer to the exponent for autocorrelation coefficients.

pDstAutoCorr Pointer to the autocorrelation coefficients vector [11].

Description

The function `ippsAutoCorr_G723` is declared in the `ippsc.h` file. This function calculates the first 11 autocorrelation coefficients of the input speech signal. This function is applied to the 180 speech samples centered on the current subframe. The functionality is as follows.

1. First, apply to speech samples the Hamming windows, given by the following formula.

$$w(i) = 0.54 - 0.46 \cos\left[\frac{(2i-1)\pi}{399}\right], i = 0, 1, \dots, 179$$

2. Next, calculate the autocorrelations as follows:

$$r(k) = \sum_{i=k}^{179} s(i) \times s(i-k), k = 0, 1, \dots, 10$$

3. Finally, add the autocorrelation with binomial window, given by:

$$b(0) = 1025 / 1024$$

$$b(i) = \exp\left[-0.5\left(\frac{2\pi f_0 i}{f_s}\right)^2\right], i = 1, \dots, 10.$$



NOTE. The function `ippsAutoCorr_G723` is actually a combination of [`ippsAutoScale`](#), [`ippsMul_NR`](#) and [`ippsAutoCorr_NormE_G723`](#) functions. The following code details the correspondence.

```
{
    int autoScale=3, corrScale;
    short vect[180];
    ippsAutoScale_16s(pSrcSpch, Vect,180, &autoScale );
    /* Apply the Hamming window */
    ippsMul_NR_16s_ISfs(HammWindow,Vect,180,15);
    /* Compute the autocorrelation coefficients */
    ippsAutoCorr_NormE_G723_16s(Vect,pDstAutoCorr,&corrScale);
    *pResultAutoCorrExp = corrScale+(autoScale<<1);
}
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcSpch</code> or <code>pDstAutoCorr</code> or <code>pResultAutoCorrExp</code> pointer is NULL.

AutoCorr_NormE_G723

Estimates normal auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorr_NormE_G723_16s(const Ipp16s* pSrc, Ipp16s* pDst, int*
    pNorm);
```

Parameters

<code>pSrc</code>	Pointer to the source vector [180].
-------------------	-------------------------------------

<i>pDst</i>	Pointer to the destination vector, which stores the estimated auto-correlation results of the source vector.
<i>pNorm</i>	Pointer to the output normalization scale factor.

Description

The function `ippAutoCorr_NormE_G723_16s` is declared in the `ippsc.h` file. This function computes 11 auto-correlation coefficients from the input signal. The first correlation coefficient (energy) is multiplied by a white noise correction factor $b(0) = 1025 / 1024$, and the remaining 10 correlation coefficients are multiplied by the binomial window coefficients $b_n, n = 1, \dots, 10$ defined in the reference C-code (see [ITU723](#)).

The auto-correlation coefficients are additionally multiplied by the factor 2^{norm0} , where $norm0 \geq 0$ is calculated so as to make the first coefficient (energy) normalized. Thus, the resulting vector *pDst* is computed as follows:

$$pDst[n] = b_n \cdot 2^{norm0} \cdot \sum_{i=0}^{179-n} pSrc[i] \cdot pSrc[i+n], 0 \leq n \leq 10.$$

The function `ippAutoCorr_NormE` stores the normalization scale factor *norm0* in *pNorm*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> or <i>pNorm</i> pointer is NULL.

LevinsonDurbin_G723

Calculates the LP coefficients from the autocorrelation coefficients.

Syntax

```
IppStatus ippLevinsonDurbin_G723_16s(const Ipp16s* pSrcAutoCorr, Ipp16s*
    pValResultSineDtct, Ipp16s* pResultResidualEnergy, Ipp16s* pDstLPC);
```

Parameters

<i>pSrcAutoCorr</i>	Pointer to the autocorrelation coefficients vector [11].
---------------------	--

pValResultSineDtct Pointer to the sine detector input/output parameter.

pResultResidualEnergy Pointer to the output residual energy, in Q15.

pDstLPC Pointer to the output LP coefficients vector [10], in Q13.

Description

The function `ippsLevinsonDurbin_G723` is declared in the `ippsc.h` file. This function calculates the 10th order LP coefficients from the autocorrelation coefficients using Levinson-Durbin algorithm. This function also performs sine detection.

To obtain LP coefficients a_i , $i = 1, 2, \dots, 10$, the following set of equations is to be solved:

$$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k), k = 1, 2, \dots, 10.$$

The function `ippsLevinsonDurbin_G723` performs the following steps:

1. Levinson-Durbin algorithm is applied to solve the above set of equations. This algorithm uses the recursion detailed in [ippsLevinsonDurbin_G729](#) function.
2. If the LPC filter used in this algorithm is unstable, that is $|k_i|$ is very close to 1.0 during recursion, just set the remaining LP coefficients to zero.
3. The sine detector parameter is updated when $i = 1$ in the recursion, as follows:

$SineDtct = SineDtct \ll 1$

If $k > 0.95$, then $SineDtct = SineDtct + 1$.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pSrcAutoCorr*, *pDstLPC*, *pValResultSineDtct*, or *pResultResidualEnergy* pointer is NULL.

LPCToLSF_G723

Converts LP coefficients to LSF coefficients.

Syntax

```
IppStatus ippsLPCToLSF_G723_16s (const Ipp16s* pSrcLPC, const Ipp16s*
    pSrcPrevLSF, Ipp16s* pDstLSF);
```

Parameters

<i>pSrcLPC</i>	Pointer to the LPC input vector [10]: a_1, \dots, a_{10} , in Q13.
<i>pSrcPrevLSF</i>	Pointer to the previous normalized LSF coefficients vector [10], in Q15.
<i>pDstLSF</i>	Pointer to the normalized LSF coefficients vector [10], in Q15.

Description

The function `ippsLPCToLSF_G723` is declared in the `ippsc.h` file. This function converts a set of 10th order LP coefficients to LSF coefficients by implementing the following operations:

1. Apply the 7.5Hz bandwidth expansion.
2. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursion:

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i)$$

$$i = 0 \dots 4,$$

$$\text{where } f_1(0) = f_2(0) = 1.0$$

3. Use Chebyshev polynomials to find the roots of $F_1(z)$ and $F_2(z)$, and obtain the LSF coefficients:

$$c_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + f_1(5)/2$$

$$c_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + f_2(5)/2$$

3. If all 10 roots needed to determine LSF coefficients are not found, use the set of previous LSF coefficients.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcLPC</i> , <i>pSrcPrevLSF</i> or <i>pDstLSF</i> pointer is NULL.

LSFToLPC_G723

Converts LSF coefficients to the LP coefficients.

Syntax

```
IppStatus ippsLSFToLPC_G723_16s(const Ipp16s* pSrcLSF, Ipp16s* pDstLPC);
IppStatus ippsLSFToLPC_G723_16s_I (Ipp16s* pSrcLSFDstLPC);
```

Parameters

<i>pSrcLSF</i>	Pointer to the input LSF coefficients vector [10], in Q15.
<i>pDstLPC</i>	Pointer to the output LP coefficients vector [10], in Q13.
<i>pSrcLSFDstLPC</i>	Pointer to the input LSF and output LP coefficients vector [10], in Q15 and Q13, respectively.

Description

The function `ippsLSFToLPC_G723` is declared in the `ippsc.h` file. This function converts a set of 10th order LSF coefficients to the LP coefficients as follows:

1. Convert LSF coefficients to the LSP coefficients as:

$$q_i = \cos(\omega_i), i = 0 \dots 9$$

2. Calculate the LP coefficients using the same algorithm as in the function [ippsLSFToLPC_G729](#) with the only difference in rounding mode required for G.723.1 bit-exact compliance.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcLSF</i> , <i>pDstLPC</i> , <i>pSrcLSFDstLPC</i> pointer is NULL.

LSFDecode_G723

Performs inverse quantization of LSFs.

Syntax

```
IppStatus ippsLSFDecode_G723_16s (const Ipp16s* quantIndex, const Ipp16s*
    pPrevLSF, int erase, Ipp16s* pQuantLSF);
```

Parameters

<i>quantIndex</i>	Pointer to the input LSP VQ indices vector [3].
<i>pPrevLSF</i>	Pointer to the input quantized LSF for previous subframes.
<i>erase</i>	Indicates frame erasure.
<i>pQuantLSF</i>	Pointer to the output quantized LSF for previous subframes.

Description

The function `ippsLSFDecode_G723_16s` is declared in the `ippsc.h` file. This function performs inverse quantization of LSFs (in other words, decodes LSP VQ indices). First three VQ table entries corresponding to the transmitted index are found. The predicted vector is added to the found vector and DC vector to form the decoded vectors in three bands separately. A stability check is performed to ensure the difference between them not is less then 31.25 Hz.

For erased frame, zero predicted vectors are chosen and stability check for the difference in 62.5 Hz is applied.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>quantIndex</i> , <i>pPrevLSF</i> or <i>pQuantLSF</i> pointer is NULL.
<code>ippStsLSFLow</code>	Indicates a warning when a stability condition is not met.

LSFQuant_G723

Quantizes LSF coefficients.

Syntax

```
IppStatus ippsLSFQuant_G723_16s32s(const Ipp16s* pSrcLSF, const Ipp16s*
    pSrcPrevLSF, Ipp32s* pResultQLSFIndex);
```

Parameters

<i>pSrcLSF</i>	Pointer to the LSF coefficients vector [10], in Q15.
<i>pSrcPrevLSF</i>	Pointer to the previous LSF coefficients vector [10], in Q15.
<i>pResultQLSFIndex</i>	Pointer to the combined index of quantized LSF coefficients. The combination is built by left-shifting the first codebook index by 16 bits and left-shifting the second codebook index by 8 bits, and then adding together the two shifted indices and the third codebook index.

Description

The function `ippsLSFQuant_G723` is declared in the `ippsc.h` file. This function quantizes the LSF coefficients to obtain the codebook indices using PSVQ. It implements the following operations:

1. Calculate the diagonal weighting matrix w , determined from the unquantized LSF coefficients, as:

$$w_{1,1} = 1 / (\omega_2 - \omega_1)$$

$$w_{10,10} = 1 / (\omega_{10} - \omega_9)$$

$$w_{j,j} = 1 / \min(\omega_j - \omega_{j-1}, \omega_{j+1} - \omega_j), j = 2 \dots 9$$

2. Calculate the prediction LSF coefficients as given by:

$$\omega_p = (\omega - \omega_{DC}) - 0.375(\omega_1 - \omega_{DC})$$

where ω are the current LSF coefficients, ω_1 are the previous LSF coefficients, and ω_{DC} are the DC LSF coefficients.

3. Search the codebook vector to minimize the following error:

$$E_1 = (\omega_p - \omega_1)^T W (\omega_p - \omega_1),$$

where ω_l is the l -th code vector in the codebook. The quantization uses 3-3-4 split.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcLSF</code> , <code>pSrcPrevLSF</code> , or <code>pResultQLSFIndex</code> pointer is NULL.

Codebook Search Functions

These functions perform open loop pitch estimation using the adaptive codebook, and search in ACELP excitation (fixed) codebook for optimal signs and positions of the pulses.

OpenLoopPitchSearch_G723

Searches for an optimal pitch value.

Syntax

```
IppStatus ippOpenLoopPitchSearch_G723_16s(const Ipp16s* pSrcWgtSpch, Ipp16s*
    pResultOpenDelay);
```

Parameters

<code>pSrcWgtSpch</code>	Pointer to the perceptually weighted speech signal. The signal length is 265, and the pointer <code>pSrcWgtSpch</code> points to its 146th element, in Q12.
<code>pResultOpenDelay</code>	Pointer to the open-loop pitch search result.

Description

The function `ippOpenLoopPitchSearch_G723` is declared in the `ippsc.h` file. This function extracts the open loop pitch from the weighted speech signal. It is applied in half-frames as follows:

1. The open loop pitch is chosen such that the cross-correlation is maximized:

$$C_{o1}(j) = \frac{\left[\sum_{i=0}^{119} s(i)s(i-j) \right]^2}{\sum_{i=0}^{119} s(i-j)s(i-j)}, \quad j = 18, \dots, 142$$

2. During the search, preference is given to smaller pitch periods to avoid choosing pitch multiples. Every found pitch value is compared with the previous best. If the difference is less than 18 and $c_{o1}(j) > c_{o1}(j')$, the process is over. Otherwise, the pitch is chosen only if $c_{o1}(j)$ is greater than $c_{o1}(j')$ by 1.25db.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcWgtSpch</code> or <code>pResultOpenDelay</code> pointer is NULL.

ACELPFixedCodebookSearch_G723

Searches the ACELP fixed codebook for the excitation.

Syntax

```
IppStatus ippSACELPFixedCodebookSearch_G723_16s(const Ipp16s* pSrcFixedCorr,
    const Ipp16s* pSrcMatrix, Ipp16s* pDstFixedSign, Ipp16s* pDstFixedPosition,
    Ipp16s* pResultGrid, Ipp16s* pDstFixedVector, Ipp16s* pSearchTimes);

IppStatus ippSACELPFixedCodebookSearch_G723_32s16s(const Ipp16s* pSrcFixedCorr,
    Ipp32s* pSrcDstMatrix, Ipp16s* pDstFixedSign, Ipp16s* pDstFixedPosition,
    Ipp16s* pResultGrid, Ipp16s* pDstFixedVector, Ipp16s* pSearchTimes);
```

Parameters

<code>pSrcFixedCorr</code>	Pointer to the correlation between residual and impulse response vector [60]
<code>pSrcMatrix</code>	Pointer to the elements of Toepliz matrix [416].
<code>pSrcDstMatrix</code>	Pointer to the input and output elements of Toepliz matrix [416].
<code>pDstFixedSign</code>	Pointer to the signs of the fixed vector [4].
<code>pDstFixedPosition</code>	Pointer to the positions of the fixed vector [4].
<code>pResultGrid</code>	Pointer to the beginning grid location.
<code>pDstFixedVector</code>	Pointer to the fixed vector [60].
<code>pSearchTimes</code>	Pointer to the input and output maximum search time

Description

The function `ippsACELPFixedCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches the ACELP fixed codebook for the excitation in the 5.3Kbps encoder. It is applied in subframes as follows:

1. There are four non-zero pulses in the fixed vector as given by:

$$c(n) = \sum_{k=0}^3 \alpha_k \delta(n - m_k) \quad , \quad n = 0, \dots, 59$$

where α_k , $k = 0..3$ and m_k , $k = 0..3$ are the signs and positions of the fixed vector respectively.

2. Search for the parameters α_k , $k = 0..3$ and m_k , $k = 0..3$, that minimize the following error:

$$\begin{aligned} \delta = & \Phi(m_0, m_0) + \Phi(m_1, m_1) + 2\alpha_0\alpha_1\Phi(m_0, m_1) + \\ & \Phi(m_2, m_2) + 2[\alpha_0\alpha_2\Phi(m_0, m_2) + \alpha_1\alpha_2\Phi(m_1, m_2)] + \\ & \Phi(m_3, m_3) + 2[\alpha_0\alpha_3\Phi(m_0, m_3) + \alpha_1\alpha_3\Phi(m_1, m_3) + \alpha_2\alpha_3\Phi(m_2, m_3)] \end{aligned}$$

where $\Phi(i, j)$ is the Toepliz matrix.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcFixedCorr</i> , <i>pSrcMatrix</i> , <i>pSrcDstMatrix</i> , <i>pSearchTimes</i> , <i>pDstFixedSign</i> , <i>pDstFixedPosition</i> , <i>pResultGrid</i> , or <i>pDstFixedVector</i> pointer is NULL.

AdaptiveCodebookSearch_G723

*Searches for the close loop pitch
and the adaptive gain index.*

Syntax

```
IppStatus ippsAdaptiveCodebookSearch_G723(Ipp16s valBaseDelay,
const Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse,
Ipp16s* pSrcPrevExcitation, const Ipp32s* pSrcPrevError,
Ipp16s* pResultCloseLag, Ipp16s* pResultAdptGainIndex, Ipp16s subFrame,
Ipp16s sineDtct, IppSpchBitRate bitRate);
```

Parameters

<i>valBaseDelay</i>	Base delay, in the range [18,145].
<i>pSrcAdptTarget</i>	Pointer to the adaptive target signal vector [60].
<i>pSrcImpulseResponse</i>	Pointer to the impulse response vector [60].
<i>pSrcPrevExcitation</i>	Pointer to the previous excitation vector [145].
<i>pSrcPrevError</i>	Pointer to the previous error vector [5].
<i>subFrame</i>	Subframe number, from 0 to 3.
<i>bitRate</i>	Transmit bit rate, equal to either IPP_SPCHBR_6300 or IPP_SPCHBR_5300.
<i>sineDtct</i>	Sine detector parameter.
<i>pResultCloseLag</i>	Pointer to the lag of close pitch.
<i>pResultAdptGainIndex</i>	Pointer to the index of the adaptive gain.

Description

The function `ippsAdaptiveCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches for the close loop pitch and the adaptive gain index. It is applied in subframes as follows:

1. For subframe 0 and 2, the close loop pitch lag is chosen in the range of [-1,1] around the appropriate open loop pitch. For subframe 1 and 3, the range is [-1, 3] around the open loop pitch.

2. Denote the close loop pitch as L_i , $i = 0...3$. The pitch predictor gains are vector quantized. For 6.3Kbps bit rate, two codebooks are used with 85 entries and 170 entries, respectively. If L_0 is less than 58 for subframe 0 and 1, or if L_2 is less than 58 for subframe 2 and 3, then the 85-entry codebook is used for pitch gain quantization. Otherwise, the 170-entry codebook is used. For 5.3Kbps bit rate, the quantization is against a 170-entry codebook.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcAdptTarget</code> , <code>pSrcImpulseResponse</code> , <code>pSrcPrevExcitation</code> , <code>pSrcPrevError</code> , <code>pResultCloseLag</code> , or <code>pResultAdptGainIndex</code> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <code>valBaseDelay</code> is not in the range [18, 145], or <code>subFrame</code> is not in the range [0, 3], or <code>bitRate</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

MPMLQFixedCodebookSearch_G723

Searches the MP-MLQ fixed codebook for the excitation.

Syntax

```
ppStatus ippMPMLQFixedCodebookSearch_G723(Ipp16s valBaseDelay, const Ipp16s*
pSrcImpulseResponse, const Ipp16s* pSrcResidualTarget, Ipp16s*
pDstFixedVector, Ipp16s* pResultGrid, Ipp16s* pResultTrainDirac, Ipp16s*
pResultAmpIndex, Ipp16s* pResultAmplitude, Ipp32s* pResultPosition, Ipp16s
subFrame);
```

Parameters

<code>valBaseDelay</code>	Base delay, in the range [18,145].
<code>pSrcImpulseResponse</code>	Pointer to the impulse response vector [60].
<code>pSrcResidualTarget</code>	Pointer to the residual target signal vector [60].
<code>pDstFixedVector</code>	Pointer to the fixed codebook vector [60].

<i>pResultGrid</i>	Pointer to the beginning grid location, 0 or 1.
<i>pResultTrainDirac</i>	Pointer to the flag to show if train Dirac function is used: 0-unused, 1-used.
<i>pResultAmpIndex</i>	Pointer to the index of the quantized amplitude.
<i>pResultAmplitude</i>	Pointer to the amplitude of the fixed codebook vector.
<i>pResultPosition</i>	Pointer to the position of the fixed codebook vector, whose amplitude has non-zero value.
<i>subFrame</i>	Subframe number, from 0 to 3.

Description

The function `ippsMPMLQFixedCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches the MP-MLQ fixed codebook for the excitation in the 6.3Kbps encoder. It is applied in subframes as follows:

1. The error is obtained by the following calculation:

$$err(n) = res(n) - G \sum_{k=0}^{M-1} \alpha_k h(n - m_k)$$

where G is the gain factor, α_k and m_k are the signs and positions of the fixed vector, respectively.

2. Search for the parameters G , α_k and m_k that minimize the mean square of the error signal $err(n)$.

3. The fixed codebook gain is obtained using the following formulae:

$$d(j) = \sum_{n=j}^{59} c(n) \cdot h(n-j)$$

$$Gm = \frac{\max_{j=0..59} \|d(j)\|}{\sum_{n=0}^{59} h(n) \cdot h(n)},$$

where $c(n)$ is the fixed vector.

Return Values

`ippsStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcImpulseResponse</i> , <i>pSrcResidualTarget</i> , <i>pDstFixedVector</i> , <i>pResultGrid</i> , <i>pResultTrainDirac</i> , <i>pResultAmpIndex</i> , <i>pResultAmplitude</i> , or <i>pResultPosition</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>subFrame</i> is not one of 0, 1, 2, or 3; or when <i>valBaseDelay</i> is not in the range [18,145].

ToeplizMatrix_G723

Calculates 416 elements of the Toepliz matrix for fixed codebook search.

Syntax

```

IppStatus ippsToeplizMatrix_G723_16s(const Ipp16s* pSrcImpulseResponse,
    Ipp16s* pDstMatrix);

IppStatus ippsToeplizMatrix_G723_16s32s(const Ipp16s* pSrcImpulseResponse,
    Ipp32s* pDstMatrix);

```

Parameters

<i>pSrcImpulseResponse</i>	Pointer to the impulse response vector [60].
<i>pDstMatrix</i>	Pointer to the elements of Toepliz matrix vector [416].

Description

The function `ippsToeplizMatrix_G723` is declared in the `ippsc.h` file. This function calculates the 416 elements in Toepliz matrix for the fixed-codebook search. Elements of the Toepliz matrix can be expressed as follows:

$$\Phi(i, j) = \sum_{n=j}^{59} h(n-i) \times h(n-j), i \leq j, 0 \leq i \leq 59,$$

where $h(i)$, $i = 0, 1, \dots, 59$, is the impulse response.

The function stores the calculated 416 elements in *pDstMatrix* in the following order:

1. $\Phi(m_i, m_i)$, ($i = 0, 1, 2, 3$), $4 \times 8 = 32$ elements; starting location: 0.

2. $\Phi(m_0, m_1)$, $8 \times 8 = 64$ elements; starting location: 32.
3. $\Phi(m_0, m_2)$, $8 \times 8 = 64$ elements; starting location: 96.
4. $\Phi(m_0, m_3)$, $8 \times 8 = 64$ elements; starting location: 160.
5. $\Phi(m_1, m_2)$, $8 \times 8 = 64$ elements; starting location: 224.
6. $\Phi(m_1, m_3)$, $8 \times 8 = 64$ elements; starting location: 288.
7. $\Phi(m_2, m_3)$, $8 \times 8 = 64$ elements; starting location: 352.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcImpulseResponse</i> or <i>pDstMatrix</i> pointer is NULL.

Gain Quantization

These functions perform fixed codebook gain estimation and gain adjustment of the postfiltered signal.

GainQuant_G723

Implements MP-MLQ gain estimation and quantization.

Syntax

```
IppStatus ippGainQuant_G723_16s (const Ipp16s* pImp, const Ipp16s* pSrc,
    Ipp16s* pDstLoc, Ipp16s* pDstAmp, Ipp32s* pMaxErr, Ipp16s* pGrid, Ipp16s*
    pAmp, int Np, int* isBest);
```

Parameters

<i>pImp</i>	Pointer to the input impulse response h vector [60].
<i>pSrc</i>	Pointer to the input target signal r vector [60].
<i>pDstLoc</i>	Pointer to the output pulse locations.
<i>pDstAmp</i>	Pointer to the output pulse amplitudes.

<i>pMaxErr</i>	Pointer to the input/output quantization error.
<i>pGrid</i>	Pointer to the output grid: 0 - if pulses are in even positions, 1 - if pulses are in odd positions.
<i>pAmp</i>	Pointer to the output index of maximum amplitude.
<i>Np</i>	Number of pulses: equal to 6 for even subframe, equal to 5 for odd subframe.
<i>isBest</i>	Indicates quantization result. Set to 0, if no pulses are found with quantization error better (less) than the input error.

Description

The function `ippsGainQuant_G723_16s` is declared in the `ippsc.h` file. This function estimates the unknown parameters, G , $\{\alpha_k\}$, $\{m_k\}$, $k = 0, \dots, Np-1$, that minimize the mean square of the error signal:

$$err[n] = r[n] - G \cdot \sum_{k=0}^{Np-1} a_k \cdot h[n - m_k]$$

The estimation is done as follows.

First, cross correlation of the impulse response and the target vector is computed:

$$d[j] = \sum_{n=j}^{59} r[n] \cdot h[n-j], \quad 0 \leq j \leq 59$$

The estimated maximum gain is computed as:

$$G_{max} = \frac{\max \{d[j]\}_{j=0..59}}{\sum_{n=0}^{59} h[n] \cdot h[n]}$$

and then is quantized by the logarithmic quantizer. The gain values selected around this quantized gain are then used to optimize the signs and locations of the pulses for both even and odd grids that yield the minimum mean square of the error signal.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pImp</code> , <code>pSrc</code> , <code>pDstLoc</code> , <code>pDstAmp</code> , <code>pMaxErr</code> , <code>pGrid</code> , <code>pAmp</code> , or <code>isBest</code> pointer is NULL.

GainControl_G723

Extracts delayed pitch contribution.

Syntax

```
IppStatus ippsgainControl_G723_16s_I (Ipp32s energy, Ipp16s* pSrcDst, Ipp16s* pGain);
```

Parameters

<code>energy</code>	The input energy coefficient.
<code>pSrcDst</code>	Pointer to the input/output post-filtered signal.
<code>pGain</code>	Pointer to the input/output gain.

Description

The function `ippsgainControl_G723_16s_I` is declared in the `ippsc.h` file. This function first computes the amplitude ratio g_s as:

$$g_s = \sqrt{\frac{\text{energy}}{\sum_{n=0}^{59} pSrcDst[n] \cdot pSrcDst[n]}}$$

If denominator in this expression is equal to 0, g_s is set to 1.

Then the input postfiltered signal is scaled as follows:

$$pSrcDst[n] = pSrcDst[n] \cdot g_n \cdot (1 + \alpha), \quad n = 0, \dots, 59,$$

where g_n is updated using the following expression, respectively:

$$g_n = (1 - \alpha) \cdot g_{n-1} + \alpha \cdot g_s, \quad n = 0, \dots, 59, \quad g_{-1} = 0$$

and α is equal to 1/16.

The output gain is equal to g_{59} .

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> or <code>pGain</code> pointer is NULL.

Filter Functions

These functions implement different types of filtering, including high pass filtering on pre-processing stage of encoding and post-processing stage of decoding, as well as the postfilter in final decode stage.

Different kinds of synthesis (IIR), harmonic, and preemphasize filter functions can be combined to perform more complex filtering and may be used in different encode and decode stages.

HighPassFilter_G723

Performs high-pass filtering of the input signal.

Syntax

```
IppStatus ippHighPassFilter_G723_16s (const Ipp16s* pSrc, Ipp16s* pDst, int* pMem) ;
```

Parameters

<code>pSrc</code>	Pointer to the source vector [240].
<code>pDst</code>	Pointer to the destination vector [240].
<code>pMem</code>	Pointer to the filter memory vector [2]. This vector must be initially filled with zeroes.

Description

The function `ippHighPassFilter_G723` is declared in the `ippsc.h` file. This function pre-processes the input signal using the following filter transfer function:

$$H_h(z) = \frac{1 - z^{-1}}{1 - \frac{127}{128}z^{-1}}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> or <i>pMem</i> pointer is NULL.

IIR16s_G723

Performs IIR filtering.

Syntax

```
IppStatus ippSIIR16s_G723_16s32s (const Ipp16s* pCoeffs, const Ipp16s* pSrc,
    Ipp32s* pDst, Ipp16s* pMem);
IppStatus ippSIIR16s_G723_16s_I (const Ipp16s* pCoeffs, Ipp16s* pSrcDst,
    Ipp16s* pMem);
IppStatus ippSIIR16s_G723_32s16s_Sfs (const Ipp16s* pCoeffs, const Ipp32s*
    pSrc, int scaleFactor, Ipp16s* pDst, Ipp16s* pMem);
```

Parameters

<i>pCoeffs</i>	Pointer to the input vector of filter coefficients vector [20]: $b_1, \dots, b_{10}, a_1, \dots, a_{10}$.
<i>pSrc</i>	Pointer to the input signal vector [60].
<i>pSrcDst</i>	Pointer to the input/output signal vector [60].
<i>scaleFactor</i>	Scale factor.
<i>pDst</i>	Pointer to the output filtered vector [60].
<i>pMem</i>	Pointer to the filter memory vector [20]. This vector must be initially filled with zeroes.

Description

These functions are declared in the `ippsc.h` file.

The functions `ippsIIR16s_G723_16s32s` and `ippsIIR16s_G723_16s_I` perform infinite impulse response (IIR) filtering using the following transfer function:

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The function `ippsIIR16s_G723_32s16s_Sfs` performs IIR filtering using the transfer function:

$$H(z) = 2^{sFs} \cdot \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The filter memory is updated.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pCoeffs</code> , <code>pSrc</code> , <code>pSrcDst</code> , <code>pDst</code> , or <code>pMem</code> pointer is NULL.

SynthesisFilter_G723

Computes the speech signal by filtering the input speech through the synthesis filter $1/A(z)$.

Syntax

```
IppStatus ippsSynthesisFilter_G723_16s32s (const Ipp16s* pLPC, const Ipp16s*
    pSrc, Ipp32s* pDst, Ipp16s* pMem);
```

```
IppStatus ippsSynthesisFilter_G723_16s (const Ipp16s* pLPC, const Ipp16s*
    pSrc, Ipp16s* pMem, Ipp16s* pDst);
```

Parameters

<i>pLPC</i>	Pointer to the input LP coefficients a_0, a_1, \dots, a_{10} , in Q11.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the filtered output.
<i>pMem</i>	Pointer to the memory supplied for filtering: short integer vector [10], initially set to zero.

Description

The function `ippsSynthesisFilter_G723` is declared in the `ippsc.h` file. This function computes the filter given by:

$$H(z) = \hat{A}^{-1}(z) = \frac{1}{a_0 + \sum_{i=1}^{10} a_i \cdot \hat{z}^{-i}}$$

This function is applied after the residual filter in computing the perceptually weighted speech signal:

$$pDst[n] = pSrc[n] \cdot pLPC[0] + \sum_{i=1}^{10} pLPC[i] \cdot pMem[n-i+1] \quad , \quad n = 0, \dots, 59$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pLPC</i> , <i>pDst</i> , or <i>pMem</i> pointer is NULL.
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

TiltCompensation_G723

Computes tilt compensation filter.

Syntax

```
IppStatus ippsTiltCompensation_G723_32s16s (Ipp16s val, const Ipp32s* pSrc,
                                             Ipp16s* pDst);
```

Parameters

<i>val</i>	The first-order partial correlation coefficient, in Q15.
<i>pSrc</i>	Pointer to the source vector [61].
<i>pDst</i>	Pointer to the output filtered vector [60].

Description

The function `ippsTiltCompensation_G723` is declared in the `ippsc.h` file. This function computes the tilt compensation filter as:

$$H_t(z) = (1 - val \cdot z^{-1})$$

$$pDst[i] = pSrc[i+1] + pSrc[i] \cdot val, \quad i = 0, \dots, 59$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.

HarmonicSearch_G723

Searches for the harmonic delay and gain for the harmonic noise shaping filter.

Syntax

```
IppStatus ippsHarmonicSearch_G723_16s(Ipp16s valOpenDelay, const Ipp16s*
                                         pSrcWgtSpch, Ipp16s* pResultHarmonicDelay, Ipp16s* pResultHarmonicGain);
```

Parameters

<i>valOpenDelay</i>	Open loop pitch, in the range [18,145].
<i>pSrcWgtSpch</i>	Pointer to the weighted speech vector [205]. The pointer points to the 146th element, in Q12.
<i>pResultHarmonicDelay</i>	Pointer to the output harmonic delay.
<i>pResultHarmonicGain</i>	Pointer to the output harmonic gain, in Q15.

Description

The function `ippsHarmonicSearch_G723` is declared in the `ippsc.h` file. This function searches for the harmonic delay and harmonic gain for the harmonic noise shaping filter, using the weighted speech and open loop pitch as input.

This function is applied in subframes as follows:

1. Find the index L to optimize the following expression:

$$C_{pw}(j) = \left[\sum_{i=0}^{59} s(i)s(i-j) \right]^2 / \left(\sum_{i=0}^{59} s(i-j)s(i-j) \right), \quad j = pitch-3..pitch+3$$

2. Calculate the optimal filter gain as follows:

$$G_{opt} = \sum_{i=0}^{59} s(i)s(i-j) / \sum_{i=0}^{59} s(i-j)s(i-j)$$

G_{opt} is saturated to [0,1].

3. Calculate the energy of the weighted speech samples and the harmonic gain:

$$E = \sum_{i=0}^{59} s^2(i)$$

$$\beta = \begin{cases} 0.3125G_{opt}, & \text{if } -10\log_{10}(1 - C_L / E) \geq 2.0 \\ 0, & \text{otherwise} \end{cases}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcWgtSpch</code> , <code>pResultHarmonicDelay</code> , or <code>pResultHarmonicGain</code> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <code>valOpenDelay</code> is not in the range [18, 145].

HarmonicNoiseSubtract_G723

Performs harmonic noise shaping.

Syntax

```
IppStatus ippHarmonicNoiseSubtract_G723_16s_I(Ipp16s val, int T, const Ipp16s* pSrc, Ipp16s* pSrcDst);
```

Parameters

<code>val</code>	The input harmonic filter coefficient, in Q15.
<code>T</code>	The input harmonic filter lag.
<code>pSrc</code>	Pointer to the input zero impulse response vector [60] of the combined filter.
<code>pSrcDst</code>	Pointer to the input/output harmonic noise weighted speech vector [60].

Description

The function `ippHarmonicNoiseSubtract_G723_16s_I` is declared in the `ippsc.h` file. This function subtracts the harmonic shaped vector `pSrc` from vector `pSrcDst`, as follows:

$$pSrcDst[n] = pSrcDst[n] - (pSrc[n] + val \cdot pSrc[n - T]) , n = 0, \dots, 59$$

This operation is used for ringing subtraction, which is performed by subtracting the zero impulse response from the harmonic weighted speech vector to obtain the target vector:

$$t[n] = w[n] - z[n]$$

Return Value

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pSrcDst</i> pointer is NULL.

DecodeAdaptiveVector_G723

Restores the adaptive codebook vector from excitation, pitch, and adaptive gain.

Syntax

```
IppStatus ippDecodeAdaptiveVector_G723_16s(Ipp16s valBaseDelay, Ipp16s
    valCloseLag, Ipp16s valAdptGainIndex, const Ipp16s* pSrcPrevExcitation,
    Ipp16s* pDstAdptVector, IppSpchBitRate bitRate);
```

Parameters

<i>valBaseDelay</i>	Base delay, in the range [18,145].
<i>valCloseLag</i>	Lag of close pitch.
<i>valAdptGainIndex</i>	Index of adaptive gain.
<i>pSrcPrevExcitation</i>	Pointer to the past excitations vector [145].
<i>bitRate</i>	Transmit bit rate, equal to either IPP_SPCHBR_6300 or IPP_SPCHBR_5300.
<i>pDstAdptVector</i>	Pointer to the adaptive codebook vector [60].

Description

The function `ippDecodeAdaptiveVector_G723` is declared in the `ippsc.h` file. This function decodes the adaptive vector from excitation, close loop pitch, and adaptive gain index. It is applied in subframes as follows:

1. Obtain the pitch as:

$$L = L_o + L_c - 1 ,$$

where L_o is the open loop pitch and L_c is the close loop pitch lag.

2. Obtain the residual from the previous excitation using the following relations:

$res(0) = exci(143-L);$

$res(1) = exci(143-L+1);$

When decoding a second subframe, the sum (L_o+L_c) can be equal to 146, so elements $exci(-2)$ and $exci(-1)$ must be available.

$res(i+2) = exci(145-L+(i\%L))$, $i = 0...61$

3. Select the adaptive codebook, G_b , from the 85-or 170-entry codebooks.

4. Obtain the adaptive vector using the formula:

$$c(i) = \sum_{j=0}^d res(i+j)G_b(20k+i) , i = 0...59$$

where k is the adaptive gain index.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcPrevExcitation</i> or <i>pDstAdptVector</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>valBaseDelay</i> is not in the range [18, 145]; or <i>valCloseLag</i> is not in the range [0, 2]; or <i>valAdptGainIndex</i> is not in the range [0, 169], or <i>bitRate</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

PitchPostFilter_G723

Calculates coefficients of the pitch post filter.

Syntax

```
IppStatus ippsPitchPostFilter_G723_16s(Ipp16s valBaseDelay, const Ipp16s*
    pSrcResidual, Ipp16s* pResultDelay, Ipp16s* pResultPitchGain, Ipp16s*
    pResultScalingGain, Ipp16s subFrame, IppSpchBitRate bitRate);
```

Parameters

<i>valBaseDelay</i>	Base delay, in the range [18,145].
<i>pSrcResidual</i>	Pointer to the residual signal vector [365]. This pointer points to the 146th element.
<i>pResultDelay</i>	Pointer to the delay of the pitch post filter.
<i>pResultPitchGain</i>	Pointer to the gain of the pitch post filter, in Q15.
<i>pResultScalingGain</i>	Pointer to the scaling gain of the pitch post filter, in Q15.
<i>subFrame</i>	Subframe number, from 0 to 3.
<i>bitRate</i>	Transmit bit rate, equal to either IPP_SPCHBR_6300 or IPP_SPCHBR_5300.

Description

The function `ippsPitchPostFilter_G723` is declared in the `ippsc.h` file. This function calculates the coefficients of the pitch post filter. It is applied in subframes as follows:

1. Search for the forward pitch lag M_f from the forward cross-correlation:

$$C_f = \sum_{i=0}^{59} res(i)res(i+M_f) , M_1 \leq M_f \leq M_2$$

where $M_1 = valBaseDelay - 3$ and $M_2 = valBaseDelay + 3$.

2. Search for the backward pitch lag M_b from the backward cross-correlation:

$$C_b = \sum_{i=0}^{59} res(i)res(i-M_b) , M_1 \leq M_b \leq M_2$$

3. If C_f or C_b is negative, then the corresponding weight and delay are set to zero.

4. Calculate the relevant residual energy as follows:

$$D_f = \sum_{i=0}^{59} res(i+M_f)res(i+M_f)$$

$$D_b = \sum_{i=0}^{59} res(i - M_b) res(i - M_b)$$

$$T_{en} = \sum_{i=0}^{59} res(i) res(i)$$

5. Obtain the optimal gain: $g = C / D$

6. Calculate the scaling gain using the following formula:

$$g_s = \sqrt{\sum_{i=0}^{59} res^2(i) / \sum_{i=0}^{59} res'^2(i)}$$

where $res'(i)$ is the post filtered residual.

7. Calculate the total gain of the post filter:

$$g_p = \gamma_{ltp} \cdot g_p \cdot g$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcResidual</code> , <code>pResultDelay</code> , <code>pResultPitchGain</code> , or <code>pResultScalingGain</code> pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>valBaseDelay</code> is not in the range [18, 145], or <code>subFrame</code> is not 0 or 3, or <code>bitRate</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

GSM-AMR Related Functions

This part of the chapter describes Intel IPP functions that can be combined to construct a bit-exact implementation of the European Telecommunications Standards Institute (ETSI) Global System for Mobile Communications (GSM) Adaptive Multi-Rate (AMR) ETSI EN 301 704 GSM 06.90 version 7.5.0 Release 2001 speech codec, more commonly known as the “GSM-AMR 06.90” codec. The primitives are primarily concerned with the well-defined, computationally expensive core operations that comprise the codec portion of the GSM-AMR system.

The GSM 06.90 AMR speech codec comprises an adaptive multi-rate algorithm that represents efficiently telephone-bandwidth digital speech using compressed data rates of 4.75, 5.15, 5.90, 6.70, 7.40, 7.95, 10.2, and 12.2 kilobits per second (kbps). The GSM-AMR system adaptively controls speech codec bit rates such that output quality is maximized for a given set of channel conditions.

Table 9-4 shows the functional grouping of the GSM-AMR primitives.

Table 9-4 Summary and Functional Groupings of the GSM-AMR Codec Primitives

Function Subset	Function Name
Basic functions	Interpolate_GSMAMR
	FFTFwd_RToPerm_GSMAMR
LP analysis	AutoCorr_GSMAMR
	LevinsonDurbin_GSMAMR
	LPCToLSP_GSMAMR
	LSPToLPC_GSMAMR
	LSFToLSP_GSMAMR
	LSPQuant_GSMAMR
	QuantLSPDecode_GSMAMR
Adaptive-codebook search	OpenLoopPitchSearchNonDTX_GSMAMR
	OpenLoopPitchSearchDTXVAD1_GSMAMR
	OpenLoopPitchSearchDTXVAD2_GSMAMR
	ImpulseResponseTarget_GSMAMR
	AdaptiveCodebookSearch_GSMAMR
	AdaptiveCodebookDecode_GSMAMR
	AdaptiveCodebookGain_GSMAMR
Fixed-codebook search	AlgebraicCodebookSearch_GSMAMR
	FixedCodebookDecode_GSMAMR

Table 9-4 Summary and Functional Groupings of the GSM-AMR Codec Primitives

Discontinuous transmission	Preemphasize_GSMAMR VAD1_GSMAMR VAD2_GSMAMR EncDTXSID_GSMAMR EncDTXHandler_GSMAMR EncDTXBuffer_GSMAMR , DecDTXBuffer_GSMAMR
Post-processing	PostFilter_GSMAMR

Sections that follow provide details on the primitives that implement each of the GSM-AMR functional blocks summarized in [Table 9-4](#), specifically:

Section [Basic Functions](#) describes some general primitives.

Section [LP Analysis and Quantization Primitives](#) describes LP analysis primitives.

Sections [Adaptive Codebook Primitives](#) and [Fixed Codebook Search](#) give details on primitives provided for the adaptive and fixed codebook search procedures, respectively.

Section [Discontinuous Transmission \(DTX\)](#) is concerned with primitives for discontinuous transmission (DTX), including functions that implement voice activity detection (VAD) algorithms 1 and 2, as well as primitives for comfort noise generation (CNG), and DTX buffer management.

Section [Post Processing](#) describes post-processing primitives.

These functions are used in the Intel IPP *GSM/AMR Speech Encoder-Decoder* sample downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Basic Functions

Interpolate_GSMAMR

Computes the weighted sum of two vectors.

Syntax

```
IppStatus ippsInterpolate_GSMAMR_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
```

Parameters

<i>pSrc1</i>	Pointer to the first input vector[<i>len</i>].
<i>pSrc2</i>	Pointer to the second input vector[<i>len</i>].
<i>pDst</i>	Pointer to the output vector[<i>len</i>].
<i>len</i>	Length of the input and output vectors.

Description

The function `ippsInterpolate_GSMAMR` is declared in `ippsc.h` file. This function computes the weighted sum of two vectors as:

$$pDst[i] = (pSrc1[i] \gg 2) + (pSrc2[i] - (pSrc2[i] \gg 2)); \quad 0 \leq i < len$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

FFTFwd_RToPerm_GSMAMR

Computes the forward fast Fourier transform (FFT) of a real signal.

Syntax

```
IppStatus ippsFFTFwd_RToPerm_GSMAMR_16s_I (Ipp16s* pSrcDst);
```

Parameters

pSrcDst Pointer to the input and output vector[128].

Description

The function `ippsFFTFwd_RToPerm_GSMAMR` is declared in `ippsc.h` file. This function computes the forward FFT of a real signal. This transform is the same as done by the general signal processing FFT function `ippsFFTFwd_RToPerm_16s_Sfs` with the following settings: `order = 7`, `flag = IPP_FFT_DIV_FWD_BY_N`, `scale factor = -1`. The result of function operation may be different because the rounding mode used in `ippsFFTFwd_RToPerm_16s_Sfs` does not guarantee a bit-exact operation of GSM AMR codec (see [Fourier Transform Functions](#) in chapter 7).

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pSrcDst* pointer is NULL.

LP Analysis and Quantization Primitives

This section describes the GSM-AMR primitives that are concerned with LP analysis, quantization, encoding, and decoding. It also provides details on primitives that accomplish the following tasks:

- Autocorrelation analysis
- Levinson-Durbin algorithm
- LPC to LSP conversion
- LSP to LPC conversion
- LSP quantization and inverse quantization
- Quantized LSP encoding and decoding.

AutoCorr_GSMAMR

Estimates the autocorrelation sequence for a block of samples.

Syntax

```
IppStatus ippsAutoCorr_GSMAMR_16s32s(const Ipp16s*  pSrcSpch, Ipp32s*
    pDstAutoCorr, IppSpchBitRate mode);
```

Parameters

<i>pSrcSpch</i>	Pointer to the input speech vector (240 samples), represented using Q15.0. This should be aligned on an 8-byte boundary.
<i>pDstAutoCorr</i>	Pointer to the autocorrelation coefficients, of length 22. For 12.2 kbps mode, elements 0 ~ 10 contain the first set of autocorrelation lags, and elements 11 ~ 21 contain the second set of autocorrelation lags. For all other modes there is only one set of autocorrelation lags contained in vector elements 0 ~ 10.
<i>mode</i>	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Description

The function `ippsAutoCorr_GSMAMR` is declared in `ippsc.h` file. This function estimates the autocorrelation sequence for a block of 240 samples (30 ms). For the 12.2 kbps mode, the 160 samples associated with the current 20 ms frame are combined with the last 80 samples from the previous frame, and two autocorrelation sequences are estimated. For all other bit rates, 160 samples from the current frame are combined with the last 40 samples from the previous frame as well as the first 40 samples from the next frame, and only one autocorrelation sequence is estimated. In particular, the estimates are formed as follows:

1. Tapered windowing – asymmetric tapered windows are applied to the input speech. Different windows are selected depending on the bit rate. For 12.2 kbps frames, unique tapered windows are applied for each of the two autocorrelation estimates, that is

$$w_1(n) = \begin{cases} 0.54 - 0.46 \cos \frac{\pi n}{159}, & n = 0, 1, \dots, 159 \\ 0.54 - 0.46 \cos \frac{(n-160)\pi}{79}, & n = 160, 161, \dots, 239 \end{cases}$$

and

$$w_2(n) = \begin{cases} 0.54 - 0.46 \cos \frac{2\pi n}{463}, & n = 0, 1, \dots, 231 \\ \cos \frac{2\pi(n-232)}{31}, & n = 232, 233, \dots, 239 \end{cases}$$

Neither w_1 nor w_2 has any look ahead. For all bit rates other than 12.2 kbps, a window centered on the current frame is applied, i.e.,

$$w_3(n) = \begin{cases} 0.54 - 0.46 \cos \frac{2\pi n}{399}, & n = 0, 1, \dots, 199 \\ \cos \frac{2\pi(n-232)}{31}, & n = 200, 201, \dots, 239 \end{cases}$$

2. Estimation of autocorrelation lags – autocorrelation lags are estimated from the windowed speech samples $s(i)$, $i = 0, 1, \dots, 239$, as follows

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k), \quad k = 0, 1, \dots, 10$$

3. Bandwidth expansion – the following binomial lag window is applied to the autocorrelation sequence(s) obtained in step 2:

$$bi(i) = \exp\left[-0.5 \times \left(\frac{2\pi f_0 i}{f_S}\right)^2\right], \quad i = 0, 1, \dots, 10$$

where $f_0 = 60$ Hz and $f_S = 8000$ Hz.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcSpch</i> or <i>pDstAutoCorr</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

LevinsonDurbin_GSMAMR

Calculates the LP coefficients using the Levinson-Durbin algorithm.

Syntax

```
IppStatus ippLevinsonDurbin_GSMAMR(const Ipp32s* pSrcAutoCorr,
    Ipp16s* pSrcDstLpc);
IppStatus ippLevinsonDurbin_GSMAMR_32s16s (const Ipp32s* pSrcAutoCorr,
    Ipp16s* pSrcDstLpc);
```

Parameters

<i>pSrcAutoCorr</i>	Pointer to the autocorrelation coefficients, a vector of length 11.
<i>pSrcDstLpc</i>	Pointer to the LP coefficients associated with the previous frame, a vector of length 11, represented using Q3.12. On output, updated to point to the LP coefficients associated with the current frame, a vector of length 11, represented using Q3.12.

Description

The function `ippsLevinsonDurbin_GSMAMR` is declared in `ippsc.h` file. This function calculates the 10th-order LP coefficients from the autocorrelation lags using the Levinson-Durbin algorithm. The detailed steps performed by the primitive are as follows:

1. Minimization of the prediction residual in the mean-square sense yields a system of linear equations that can be solved efficiently using the Levinson-Durbin algorithm to yield a set of LP coefficients, a_i , $i = 1, 2, \dots, 10$, that is

$$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k), \quad k = 1, 2, \dots, 10.$$

2. The Levinson-Durbin algorithm performs the following recursion to obtain the prediction coefficients:

$$E^{[0]} = r(0)$$

for $i = 1$ to 10

$$a_0^{[i-1]} = 1$$

$$k_i = - \left[\sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j) \right] / E^{[i-1]}$$

$$a_i^{[i]} = k_i$$

for $j = 1$ to $i-1$

$$a_j^{[i]} = a_j^{[i-1]} + k_i \times a_{i-j}^{[i-1]}$$

end

$$E^{[i]} = E^{[i-1]} - k_i^2 E^{[i-1]}$$

end

3. Unstable synthesis filter exception handling: if autocorrelation analysis yields an unstable LPC synthesis filter, i.e., if $|k_i|$ is close to or larger than 1.0, then the LP coefficients associated with the previous frame should replace the LP coefficients estimated on the current frame.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcAutoCorr</code> or <code>pSrcDstLpc</code> pointer is NULL.

LPCToLSP_GSMAMR

Converts LP coefficients to line spectrum pairs.

Syntax

```
IppStatus ipp2LPCToLSP_GSMAMR_16s(const Ipp16s* pSrcLpc, const Ipp16s*
    pSrcPrevLsp, Ipp16s* pDstLsp);
```

Parameters

<code>pSrcLpc</code>	Pointer to the eleven-element LP coefficient vector, represented using Q3.12.
<code>pSrcPrevLsp</code>	Pointer to the ten-element LSP coefficient vector associated with the previous frame, represented using Q0.15.
<code>pDstLsp</code>	Pointer to the ten-element LSP coefficient vector, represented using Q0.15.

Description

The function `ipp2LPCToLSP_GSMAMR` is declared in `ippsc.h` file. This function converts a set of 10th-order LP coefficients to an equivalent set of line spectrum pairs (LSPs). The functionality is as follows:

1. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$ using the recursive relations

$$f_1(i+1) = a_{i+1} + a_{i-1} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{i-1} + f_1(i) \quad i = 0, 1, \dots, 4$$

where $f_1(0) = f_2(0) = 1.0$.

2. Use Chebyshev polynomials to evaluate $F_1(z)$ and $F_2(z)$. The Chebyshev polynomials are given by the following formulas;

$$C_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + \frac{f_1(5)}{2}$$

$$C_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + \frac{f_2(5)}{2}$$

3. Evaluate $F_1(z)$ and $F_2(z)$ on 60 points equally spaced between 0 and π , checking for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided 4 times to track the root.
4. If 10 roots for LSP coefficients are not found during the search, the previous set of LSPs is used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcLpc</i> , <i>pSrcPrevLsp</i> or <i>pDstLsp</i> pointer is NULL.

LSPToLPC_GSMAMR

Converts LSPs to LP coefficients.

Syntax

```
IppStatus ippLSPToLPC_GSMAMR_16s(const Ipp16s* pSrcLsp, Ipp16s* pDstLpc);
```

Parameters

<i>pSrcLsp</i>	Pointer to the ten-element LSP coefficient vector, represented using Q0.15.
----------------	---

pDstLpc Pointer to the eleven-element LP coefficient vector, represented using Q3.12.

Description

The function `ippsLSPToLPC_GSMAMR` is declared in `ippsc.h` file. This function converts a set of 10th-order LSPs to LP coefficients. The functionality is as follows:

1. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursive relations for $i = 1$ to 5

$$f_1(i) = -2q_{2i-1} \times f_1(i-1) + 2f_1(i-2)$$

for $j = i-1$ down to 1

$$f_1(j) = f_1(j) - 2q_{2i-1} \times f_1(j-1) + 2f_1(j-2)$$

end

end

where initial values $f_1(0) = 1$ and $f_1(-1) = 0$. The coefficients $f_2(i)$ are computed similarly by replacing q_{2i-1} by q_{2i} .

2. Multiply $F_1(z)$ and $F_2(z)$ by $1+z^{-1}$ and $1-z^{-1}$, respectively, to obtain $F_1'(z)$ and $F_2'(z)$ using the following relations:

$$f_1'(i) = f_1(i) + f_1(i-1), \quad i = 1, 2, \dots, 5$$

$$f_2'(i) = f_2(i) + f_2(i-1), \quad i = 1, 2, \dots, 5$$

3. Compute the LP coefficients from the polynomials $F_1'(z)$ and $F_2'(z)$ as follows

$$a_i = \begin{cases} 0.5 f_1'(i) + 0.5 f_2'(i), & i = 1, 2, \dots, 5 \\ 0.5 f_1'(11-i) + 0.5 f_2'(11-i), & i = 6, 7, \dots, 11 \end{cases}$$

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pSrcLsp* or *pDstLpc* pointer is NULL.

LSFToLSP_GSMAMR

Converts Line Spectral Frequencies to LSP coefficients.

Syntax

```
IppStatus ippsLSFToLSP_GSMAMR_16s (const Ipp16s* pLSF, Ipp16s* pLSP) ;
```

Parameters

pLSF Pointer to the vector [10] of LSF normalized by factor $1/\pi$ in Q14, so given in the range [0,1].

pLSP Pointer to the LSP vector [10], in Q15 in the range [-1,1].

Description

The function `ippsLSFToLSP_GSMAMR` is declared in `ippsc.h` file. This function converts the Line Spectral Frequencies (LSF) to the LSP coefficients as given by:

$$lsp_n = \cos(lsf_n), 1 \leq n \leq 10$$

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pLSF* or *pLSP* pointer is NULL.

LSPQuant_GSMAMR

Quantizes the LSP coefficient vector.

Syntax

```
IppStatus ippsLSPQuant_GSMAMR_16s(const Ipp16s* pSrcLsp, Ipp16s*
    pSrcDstPrevQLsfResidual, Ipp16s* pDstQLsp, Ipp16s* pDstQLspIndex,
    IppSpchBitRate mode);
```


Parameters

<i>pSrcLsp</i>	Pointer to the unquantized 20-element LSP vector, represented using Q0.15. For 12.2 kbps frames, the first LSP set is contained in vector elements 0 ~ 9, and the second LSP set is contained in vector elements 10 ~ 19. For all other bit rates, only elements 0 to 9 are valid and used for the quantization.
<i>pSrcDstPrevQLsfResidual</i>	Pointer to the ten-element quantized LSF residual from the previous frame, represented using Q0.15. On output, points to the ten-element quantized LSF residual for the current frame, represented using Q0.15.
<i>pDstQLsp</i>	Pointer to the 20-element quantized LSP vector, represented using Q0.15. For 12.2 kbps frames, elements 0 to 9 contain the first quantized LSP set, and elements 10 to 19 contain the second quantized LSP set. For all other bit rates there is only one LSP set contained in elements 0 to 9.
<i>pDstQLspIndex</i>	Pointer to the five-element vector of quantized LSP indices. For 12.2Kbps frames, all five elements contain valid data; for all other bit rates, only the first three elements contain valid indices (see the following discussion of SMQ and SVQ for the various modes).
<i>mode</i>	Bit rate specifier. The enumerated values of IPP_SPCHBR_4750 to IPP_SPCHBR_12200 are valid.

Description

The function `ippsLSPQuant_GSMAMR` is declared in `ippsc.h` file. This function quantizes the LSP coefficient vector, then obtains quantized LSP codebook indices. The functionality can be summarized as follows:

1. LSP-to-LSF conversion – the LSPs are mapped to line spectrum frequencies (LSFs). For 12.2 kbps frames, two sets of LSPs are converted to LSFs. For all other bit rates, a single set of LSPs is converted using the following relation:

$$f_i = \frac{f_s}{2\pi} \arccos(q_i)$$

2. LSF prediction – first-order moving average (MA) prediction is applied to the LSF vectors. Then, the prediction residual is quantized. For 12.2 kbps frames, two prediction residual vectors $r^{(1)}(n)$ and $r^{(2)}(n)$ are formed as follows:

$$r^{(1)}(n) = z^{(1)}(n) - 0.65\hat{r}^{(1)}(n-1), \quad r^{(2)}(n) = z^{(2)}(n) - 0.65\hat{r}^{(2)}(n-1)$$

where $z^{(1)}(n)$ and $z^{(2)}(n)$ are the zero-mean LSF vectors at frame n and $\hat{r}^{(2)}(n-1)$ is the quantized residual vector from frame $n-1$.

For all other bit rates, a signal prediction residual vector, $r(n)$, is given by:

$$r_j(n) = z_j(n) - \alpha_j \hat{r}_j(n-1), \quad j = 0, \dots, 9$$

where $z(n)$ is the mean-removed LSF vector at frame n , and $\hat{r}(n-1)$ is the quantized residual vector from frame $n-1$.

3. Quantization – Split matrix quantization (SMQ) is applied to the prediction residual vectors. For 12.2 kbps frames, the matrix $(r^{(1)}, r^{(2)})$ is split into 5 submatrices of dimension 2x2, and the 5 submatrices are quantized with 7, 8, 9, 8, and 6 bits, respectively. For all other bit rates, the vector r is split into 3 subvectors of dimension 3, 3, and 4. The three subvectors are quantized with 7-9 bits. The VQ search identifies the index k which minimizes the weighted error

$$E_{LSP} = \sum_{i=0}^9 [r_i w_i - \hat{r}_i^k w_i]^2$$

where the weighting factor, w_i , is given by

$$w_i = \begin{cases} 3.347 - \frac{1.547}{450} d_i, & d_i < 450 \\ 1.8 - \frac{0.8}{1050} (d_i - 450), & \text{otherwise} \end{cases}$$

and $d_i = f_{i+1} - f_{i-1}$ with $f_0 = 0$, and $f_{11} = 4000$.

4. Resonance minimization – The quantized LSF vector elements are rearranged in order to avoid sharp resonances in the LP synthesis filter.
5. LSF-to-LSP conversion – convert the quantized LSFs to LSPs.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcLsp</code> , <code>pSrcDstPrevQLsfResidual</code> , <code>pDstQLsp</code> , or <code>pDstQLspIndex</code> pointer is NULL.

<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
-----------------------------	---

QuantLSPDecode_GSMAMR

Decodes quantized LSPs.

Syntax

```
IppStatus ippsQuantLSPDecode_GSMAMR_16s(const Ipp16s* pSrcQLspIndex, Ipp16s*
    pSrcDstPrevQLsfResidual, Ipp16s* pSrcDstPrevQLsf, Ipp16s* pSrcDstPrevQLsp,
    Ipp16s* pDstQLsp, Ipp16s bfi, IppSpchBitRate mode);
```

Parameters

<i>pSrcQLspIndex</i>	Pointer to the five-element vector containing codebook indices of the quantized LSPs. For 12.2 kbps frames, all five elements contain valid indices; for all other bit rates, only the first three elements contain valid indices.
<i>pSrcDstPrevQLsfResidual</i>	Pointer to the ten-element quantized LSF residual from the previous frame, represented using Q0.15. On output, points to the ten-element updated LSF residual, represented using Q0.15.
<i>pSrcDstPrevQLsf</i>	Pointer to the ten-element quantized LSF vector from the previous frame, represented using Q0.15. On output, points to the ten-element updated quantized LSF vector, represented using Q0.15.
<i>pSrcDstPrevQLsp</i>	Pointer to the ten-element quantized LSP vector from the previous frame, represented using Q0.15. On output, points to the ten-element updated quantized LSP vector, represented using Q0.15.
<i>pDstQLsp</i>	Pointer to a 40-element vector containing four subframe LSP sets. Two sets are generated by interpolation for 12.2 kbps frames; for all other bit rates three sets are generated by interpolation. All elements are represented in using Q0.15.

<i>bfi</i>	Bad frame indicator; “0” signifies a good frame; all other values signify a bad frame.
<i>mode</i>	Bit rate specifier. The enumerated values of IPP_SPCHBR_4750 to IPP_SPCHBR_12200 are valid.

Description

The function `ippQuantLSPDecode_GSMAMR` is declared in `ippsc.h` file. This function decodes quantized LSPs from the received codebook index if the errors are not detected on the received frame. Otherwise, the function recovers the quantized LSPs from previous quantized LSPs using linear interpolation. The functionality can be summarized as follows:

1. If no errors are detected on the current frame, obtain the quantized LSPs from the codebook indices and the previous quantized residual using inverse LSP quantization.
2. If errors are detected on the current frame, quantized LSFs are obtained using the following rate-dependent interpolation scheme:

$$\begin{cases} lsf_q1(i) = lsf_q2(i) = \alpha \times past_lsf_q(i) + (1 - \alpha) \times mean_lsf(i), & \text{12.2kbit/s mode} \\ lsf_q(i) = \alpha \times past_lsf_q(i) + (1 - \alpha) \times mean_lsf(i), & \text{otherwise} \end{cases}$$

where $i = 0, 1, \dots, 9$, $\alpha = 0.95$, lsf_q1 and lsf_q2 (for 12.2 kbps) are two sets of quantized LSF vectors for current frame, $past_lsf_q$ is lsf_q2 of the previous frame, and $mean_lsf$ is the average LSF vector. Note that there is only one set of quantized LSF coefficients for rates other than 12.2 kbps. The corresponding quantized LSPs are obtained by LSF-to-LSP conversion.

3. Linear interpolation is applied to generate four sets of quantized LSPs from the decoded set(s) of LSPs and the quantized LSPs from the previous frame.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcQLspIndex</i> , <i>pSrcDstPrevQLsfResidual</i> , <i>pSrcDstPrevQLsf</i> , <i>pSrcDstPrevQLsp</i> or <i>pDstQLsp</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

Adaptive Codebook Primitives

This section describes primitives that are concerned with various aspects of the adaptive codebook, including primitives that perform the following functions:

- Open-loop pitch search, including Non-DTX, VAD1, and VAD2
- Impulse response and target signal computation
- Adaptive codebook search
- Decoding of the adaptive codebook vector.

Open-Loop Pitch Search (OLP)

Three functions are provided for OLP search. Use of these functions is mutually exclusive, that is, only one is appropriate for an application during any given frame type. The appropriate choice of an OLP search function depends upon the state of the DTX and VAD modules. The OLP search functions should be applied as follows:

Encoder Mode	Appropriate Function
DTX disabled	<code>ippsOpenLoopPitchSearchNonDTX_GSMAMR</code>
DTX, VAD 1 enabled	<code>ippsOpenLoopPitchSearchDTXVAD1_GSMAMR</code>
DTX, VAD 2 enabled	<code>ippsOpenLoopPitchSearchDTXVAD2_GSMAMR</code>

The OLP search functions extract a pitch estimate from a weighted version of the input speech. For 5.15 and 4.75 kbps frames, the search is performed once per frame. For all other modes, the search is performed twice per frame. A unique search is employed for 10.2 kbps frames. The OLP search details are as follows:

1. If the transmission bit rate is 10.2 kbps,
 - a. Compute a windowed autocorrelation of weighted speech, that is,

$$R(k) = w(k) \times \sum_{n=0}^{79} sw(n) \times sw(n-k), \quad 20 \leq k \leq 143$$

where the sequence $sw(n)$ contains weighted speech, and w is the weighting function given by

$$w(k) = wl(k) \times wn(k)$$

In this weighting function, wl emphasizes low pitch, and is defined in terms of the table cw , while wn is a sequence of neighboring emphasis lag coefficients associated with the previous frame, i.e.

$$wn(k) = \begin{cases} cw(|T_{old} - k| + 20) & weighflag = 1 \\ 1 & \text{otherwise} \end{cases}$$

and the parameter T_{old} is the median filtered pitch lag of 5 previous voiced speech half frames.

The estimated open-loop pitch lag is the value k that maximizes $R(k)$. It is denoted by T_{op} .

- b. Compute the optimal open-loop gain using the relation

$$g = \frac{\sum_{n=0}^{79} sw(n) \times sw(n - T_{op})}{\sum_{n=0}^{79} sw(n) \times sw(n)} - 0.4$$

In addition,

$$v = \begin{cases} 1 & g > 0 \\ 0.9v & \text{otherwise} \end{cases}$$

and

$$weighflag = \begin{cases} 1 & v > 0 \\ 0 & \text{otherwise} \end{cases}$$

If $g > 0$, the previous pitch lag buffer and v median pitch lag of the previous pitch lags are updated.

2. For rates other than 10.2 kbps, the following OLP search procedure is employed:

- a. Three maxima are found in three different ranges for the correlations given by

$$R(k) = \sum_{n=0}^{length-1} sw(n) \times sw(n-k)$$

For 5.15 and 4.75 kbps frames, $length = 160$. For all other bit rates (except 10.2 kbps), $length = 80$.

- b. Normalize the three maximum correlations according to

$$M_i = \frac{M_i}{\sqrt{\sum_{n=0}^{length-1} sw^2(n - T_i)}} \quad i=1,2,3$$

- c. Determine the best open-loop lag using the following rule:

$$T_{op} = T_1, M(T_{op}) = M_1$$

$$\text{if}(M_2 > 0.85 M(T_{op}))$$

$$M(T_{op}) = M_2, \text{ and } T_{op} = T_2$$

$$\text{if}(M_3 > 0.85 M(T_{op}))$$

$$T_{op} = T_3$$

Each of the OLP search functions implements the rate-dependent search algorithms described above. Next, details are given for non-DTX, VAD1, and VAD2 OLP search primitives.

OpenLoopPitchSearchNonDTX_GSMAMR

Computes the open-loop pitch lag.

Syntax

```
IppStatus ippsOpenLoopPitchSearchNonDTX_GSMAMR_16s(const Ipp16s* pSrcWgtLpc1,
const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcSpch,
Ipp16s* pValResultPrevMidPitchLag, Ipp16s* pValResultVvalue,
Ipp16s* pSrcDstPrevPitchLag, Ipp16s* pSrcDstPrevWgtSpch,
Ipp16s* pDstOpenLoopLag, Ipp16s* pDstOpenLoopGain, IppSpchBitRate mode);
```

Parameters

<i>pSrcWgtLpc1</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. This set of LP coefficients comprises the numerator coefficients for the perceptual weighting filter.
<i>pSrcWgtLpc2</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. This set of LP coefficients comprises the denominator coefficients for the perceptual weighting filter.
<i>pSrcSpch</i>	Pointer to the 170-sample input speech vector, represented using Q15.0.

<i>pValResultPrevMidPitchLag</i>	Pointer to the median filtered pitch lag of the 5 previous voiced speech half-frames, represented using Q15.0. On output, points to the updated median filtered pitch lag vector (5 previous voiced speech half-frames), represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>pValResultVvalue</i>	Pointer to the adaptive parameter <i>v</i> described in Open-Loop Pitch Search (OLP) , represented using Q0.15. Used also as an output argument. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevPitchLag</i>	Pointer to the five-element vector that contains the pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five previous voiced half-frames. This argument is used only for 10.2 kbps frames.
<i>pSrcDstPrevWgtSpch</i>	Pointer to a 143-element vector containing perceptually weighted speech from the previous frame, represented using Q15.0. On output, points to the updated 143-element vector of perceptually weighted speech, represented using Q15.0.
<i>pDstOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
<i>pDstOpenLoopGain</i>	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>mode</i>	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Description

The function `ippsOpenLoopPitchSearchNonDTX_GSMAMR` is declared in `ippsc.h` file. This function computes the open-loop pitch lag (as well as optimal pitch gain for 10.2 kbps frames only) when both DTX and VAD are disabled. The search algorithm is described in [Open-Loop Pitch Search \(OLP\)](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

OpenLoopPitchSearchDTXVAD1_GSMAMR

*Extracts an open-loop pitch lag estimate
(VAD 1 scheme is enabled).*

Syntax

```
IppStatus ippOpenLoopPitchSearchDTXVAD1_GSMAMR_16s(const Ipp16s* pSrcWgtLpc1,  
const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcSpch,  
Ipp16s* pValResultToneFlag, Ipp16s* pValResultPrevMidPitchLag,  
Ipp16s* pValResultVvalue, Ipp16s* pSrcDstPrevPitchLag,  
Ipp16s* pSrcDstPrevWgtSpch, Ipp16s* pResultMaxHpCorr, Ipp16s*  
pDstOpenLoopLag, Ipp16s* pDstOpenLoopGain, IppSpchBitRate mode);
```

Parameters

<code>pSrcWgtLpc1</code>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the numerator of the perceptual weighting filter.
<code>pSrcWgtLpc2</code>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the denominator of the perceptual weighting filter.
<code>pSrcSpch</code>	Pointer to the 170-element input speech vector, represented using Q15.0.

<i>pValResultToneFlag</i>	Pointer to the tone flag for the VAD module. On output, points to the updated tone flag for the VAD module.
<i>pValResultPrevMidPitchLag</i>	Pointer to a vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. On output, points to the updated vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>pValResultVvalue</i>	Pointer to the adaptive parameter <i>v</i> , represented using Q0.15. On output, points to the updated adaptive parameter <i>v</i> , represented using Q0.15. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevPitchLag</i>	Pointer to a five-element vector of pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five most recent voiced speech half-frames. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevWgtSpch</i>	Pointer to a 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0. On output, points to the updated 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0.
<i>pResultMaxHpCorr</i>	Pointer to the correlation maximum.
<i>pDstOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
<i>pDstOpenLoopGain</i>	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>mode</i>	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Description

The function `ippsOpenLoopPitchSearchDTXVAD1_GSMAMR` is declared in `ippsc.h` file. This function extracts an open-loop pitch lag estimate from the weighted input speech when the VAD 1 scheme is enabled using a version of the pitch search algorithm described in [Open-Loop Pitch Search \(OLP\)](#), that is modified as follows:

1. For 10.2 kbps frames, the following modification is applied after the best open-loop pitch is found:

- a. Update the tone flag (when initialized or reset, it is set to 0) in the following way

$$toneflag \gg 1$$

$$DelayEnergy = \sum_{n=0}^{79} sw^2(n - T_{op})$$

$$MaxCorr = \sum_{n=0}^{79} sw(n - T_{op}) \times sw(n)$$

$$\text{if } (0.325 \times DelayEnergy < MaxCorr) \quad toneflag = toneflag | 0x4000$$

- b. On the second OLP search for the current frame, find the maximum of the high-pass filtered autocorrelations, i.e.,

$$maxhpcorr = \max(R(k) \times 2 - R(k-1) - R(k+1) | k = 142, \dots, 24)$$

Then, $maxhpcorr$ is normalized by $NormFactor = frameEnergy - frameCorr$:

$$frameEnergy = \sum_{n=0}^{79} sw^2(n) \quad , \quad frameCorr = \sum_{n=0}^{79} sw(n) \times sw(n-1)$$

2. For all other bit rates, the following modifications are applied:

- a. Before the open-loop pitch search, update the tone flag as follows:

$$toneflag = toneflag \gg 1$$

If the bit rate is either 5.15 or 4.75 kbps, update the tone flag as follows:

$$toneflag = toneflag \gg 1, \quad toneflag = toneflag | 0x2000$$

- b. After finding three open-loop pitch candidates, update the tone flag as follows:

$$\text{if } (DelayEnergy \times 0.65 < MaxCorr) \quad toneflag = toneflag | 0x4000$$

This update is repeated three times with *DelayEnergy* and *MaxCorr* corresponding to the three pitch candidates. Note that the computation length of *DelayEnergy* and *MaxCorr* for 4.75 and 5.15 kbps frames is 160 samples. For all other bit rates, the length is 80 samples.

- c. On the second OLP search for each frame, find the maximum of the high passed autocorrelations. The implementation is identical the 10.2 kbps correlation search, but the search range is rate-dependent.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

OpenLoopPitchSearchDTXVAD2_GSMAMR

*Extracts an open-loop pitch lag estimate
(VAD 2 scheme is enabled).*

Syntax

```
IppStatus ippsOpenLoopPitchSearchDTXVAD2_GSMAMR(const Ipp16s* pSrcWgtLpc1,
const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcSpch,
Ipp16s* pValResultPrevMidPitchLag, Ipp16s* pValResultVvalue,
Ipp16s* pSrcDstPrevPitchLag, Ipp16s* pSrcDstPrevWgtSpch,
Ipp32s* pResultMaxCorr, Ipp32s pResultWgtEnergy, Ipp16s* pDstOpenLoopLag,
Ipp16s* pDstOpenLoopGain, IppSpchBitRate mode);

IppStatus ippsOpenLoopPitchSearchDTXVAD2_GSMAMR_16s32s(const Ipp16s*
pSrcWgtLpc1, const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcSpch, Ipp16s*
pValResultPrevMidPitchLag, Ipp16s* pValResultVvalue, Ipp16s*
pSrcDstPrevPitchLag, Ipp16s* pSrcDstPrevWgtSpch, Ipp32s* pResultMaxCorr,
Ipp32s pResultWgtEnergy, Ipp16s* pDstOpenLoopLag, Ipp16s*
pDstOpenLoopGain, IppSpchBitRate mode);
```

Parameters

<i>pSrcWgtLpc1</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the numerator of the perceptual weighting filter.
<i>pSrcWgtLpc2</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the denominator of the perceptual weighting filter.
<i>pSrcSpch</i>	Pointer to the 170-element input speech vector, represented using Q15.0.
<i>pValResultToneFlag</i>	Pointer to the tone flag for the VAD module.
<i>pValResultPrevMidPitchLag</i>	Pointer to a vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. On output, points to the updated vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>pValResultVvalue</i>	Pointer to the adaptive parameter v , represented using Q0.15. On output, points to the updated adaptive parameter v , represented using Q0.15. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevPitchLag</i>	Pointer to a five-element vector of pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five most recent voiced speech half-frames. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevWgtSpch</i>	Pointer to a 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0. On output, points to the updated 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0.
<i>pResultMaxCorr</i>	Pointer to the correlation maximum.

<i>pResultWgtEnergy</i>	Pointer to the pitch delayed energy of the weighted speech signal, as described above. The output may be scaled, and the Q representation is not given.
<i>pDstOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
<i>pDstOpenLoopGain</i>	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>mode</i>	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Description

The function `ippsOpenLoopPitchSearchDTXVAD2_GSMAMR` is declared in `ippsc.h` file. This function extracts an open-loop pitch lag estimate from the weighted input speech when the VAD 2 scheme is enabled, using a version of the pitch search algorithm described in [Open-Loop Pitch Search \(OLP\)](#), that is modified in the following way:

After finding the best open-loop pitch, extract from the weighted speech the maximum correlation *MaxCorr* and the delayed energy *DelayEnergy*. For both 4.75 and 5.15 kbps frames, the computation is carried for 160 samples. For all other bit rates, the computation is carried for only 80 samples using the relations

$$MaxCorr = \sum_{n=0}^{length-1} sw(n) \times sw(n - T_{op})$$

$$DelayEnergy = \sum_{n=0}^{length-1} sw^2(n - T_{op})$$

The *MaxCorr* and *DelayEnergy* values corresponding to the two half-frame open-loop pitch lag estimates are combined to obtain whole-frame estimates of *MaxCorr* and *DelayEnergy* (except during 4.75 and 5.15 kbps frames, when the OLP search is performed only once per frame).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

ImpulseResponseTarget_GSMAMR

Computes the impulse response and target signal required for the adaptive codebook search.

Syntax

```
IppStatus ippImpulseResponseTarget_GSMAMR_16s(const Ipp16s* pSrcSpch, const
Ipp16s* pSrcWgtLpc1, const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcQLpc,
const Ipp16s* pSrcSynFltState, const Ipp16s* pSrcWgtFltState, Ipp16s*
pDstImpulseResponse, Ipp16s* pDstLpResidual, Ipp16s* pDstAdptTarget);
```

Parameters

<code>pSrcSpch</code>	Pointer to the 50-element input speech vector, where elements 0 - 9 are from the previous subframe, and elements 10 - 49 are from the current subframe.
<code>pSrcWgtLpc1</code>	Pointer to an eleven-element vector of weighted LP coefficients associated with $A(z/\gamma_1)$ on the current subframe, represented using Q3.12.
<code>pSrcWgtLpc2</code>	Pointer to an eleven-element vector of weighted LP coefficients associated with $A(z/\gamma_2)$ on the current subframe, represented using Q3.12.
<code>pSrcQLpc</code>	Pointer to an eleven-element vector of quantized LP coefficients for the current subframe, represented using Q3.12.
<code>pSrcSynFltState</code>	Pointer to the ten-element vector that contains the state of the synthesis filter, represented using Q15.0.

<i>pSrcWgtFltState</i>	Pointer to the ten-element vector that contains the state of the weighting filter, represented using Q15.0.
<i>pDstImpulseResponse</i>	Pointer to the 40-element vector that contains the impulse response, represented using Q3.12.
<i>pDstLpResidual</i>	Pointer to the 40-element vector that contains the LP residual, represented using Q15.0.
<i>pDstAdptTarget</i>	Pointer to the 40-element vector that contains the adaptive codebook search target signal, represented using Q15.0.

Description

The function `ippsImpulseResponseTarget_GSMAMR` is declared in `ippsc.h` file. This function computes the impulse response and target signal required for the adaptive codebook search. This function is performed on a subframe basis using the following approach:

1. The impulse response $h(n)$ of the weighted synthesis filter,

$$H(z)W(z) = A(z/\gamma_1) / [\hat{A}(z)A(z/\gamma_2)] \quad ,$$

is computed by applying the filters $1/\hat{A}(z)$ and $1/A(z/\gamma_2)$ to the zero-padded impulse response of the filter $A(z/\gamma_1)$.

2. The target signal is then obtained by applying to the LP residual $res_{LP}(n)$ the cascaded synthesis and weighting filters, $1/\hat{A}(z)$, and $A(z/\gamma_1) / A(z/\gamma_2)$, respectively. The adaptive codebook search also uses the residual signal $res_{LP}(n)$ to update the history of past excitations. The LP residual is obtained by inverse filtering the input speech, i.e.,

$$res_{LP}(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i s(n-i)$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcSpch</i> , <i>pSrcWgtLpc1</i> , <i>pSrcWgtLpc2</i> , <i>pSrcQLpc</i> , <i>pSrcSynFltState</i> , <i>pSrcWgtFltState</i> , <i>pDstImpulseResponse</i> , <i>pDstLpResidual</i> , or <i>pDstAdptTarget</i> pointer is NULL.

AdaptiveCodebookSearch_GSMAMR

Performs the adaptive codebook search.

Syntax

```
IppStatus ippsAdaptiveCodebookSearch_GSMAMR_16s(const Ipp16s* pSrcTarget,
const Ipp16s* pSrcImpulseResponse, Ipp16s* pSrcOpenLoopLag,
Ipp16s* pValResultPrevIntPitchLag, Ipp16s* pSrcDstExcitation,
Ipp16s* pResultFracPitchLag, Ipp16s* pResultAdptIndex,
Ipp16s* pDstAdptVector, Ipp16s subFrame, IppSpchBitRate mode);
```

Parameters

<i>pSrcTarget</i>	Pointer to the 40-element adaptive target signal vector, represented using Q15.0. This should be aligned on an 8-byte boundary.
<i>pSrcImpulseResponse</i>	Pointer to the 40-element impulse response of the weighted synthesis filter, represented using Q3.12. This should be aligned on an 8-byte boundary.
<i>pSrcOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
<i>pValResultPrevIntPitchLag</i>	Pointer to the previous integral pitch lag.
<i>pSrcDstExcitation</i>	Pointer to the 194-element excitation vector. Elements 0 ~ 153 contain samples of the past excitation, represented using Q15.0. Elements 154 ~ 193 contain samples of the prediction residual, however, the prediction residuals are used only when the subframe length exceeds the integer closed-loop pitch estimate.
<i>pValResultPrevIntPitchLag</i>	Pointer to the current integer pitch lag.
<i>pSrcDstExcitation</i>	Pointer to the updated 194-element excitation vector. Elements 0 ~ 153 contain past excitations, and are represented using Q15.0. Elements 154 ~ 193 contain the 40-sample adaptive codebook vector, <i>v</i> .

<i>pResultFracPitchLag</i>	Pointer to the fractional pitch lag obtained during the adaptive codebook search.
<i>pResultAdptIndex</i>	Pointer to the coded closed-loop pitch index.
<i>pDstAdptVector</i>	Pointer to the 40-sample adaptive codebook vector, represented using Q15.0.
<i>subFrame</i>	Subframe index.
<i>mode</i>	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Description

The function `ippsAdaptiveCodebookSearch_GSMAMR` is declared in `ippsc.h` file. This function performs the adaptive codebook search. The adaptive codebook search consists of a closed-loop pitch search followed by computation of an adaptive excitation vector. The adaptive excitation vector is obtained by interpolating the past excitation at the fractional pitch lag obtained during the closed-loop pitch search. The adaptive codebook is searched on every subframe. A detailed description of the adaptive codebook search procedure is given next:

1. A closed-loop pitch analysis is performed in the neighborhood of the open-loop pitch estimate, T_{op} , on a subframe basis. In the first and third subframes (only the first subframe for 5.15 and 4.75 kbps modes), the search neighborhood is rate-dependent. For 12.2 kbps frames, the search range is $T_{op} \pm 3$, and is bounded by 18...143. For 5.15 or 4.75 kbps frames, the search range is $T_{op} \pm 5$, and is bounded by 20...143. For all other bit rates, the search range is $T_{op} \pm 3$, and is bounded by 20...143. For the second and fourth subframes, the search neighborhood surrounds T_I , the nearest integer to the fractional pitch lag of the previous subframe, and the neighborhood boundaries are also rate-dependent. For 12.2 kbps frames, the search range is $[T_I - 5 \frac{3}{6}, T_I + 4 \frac{3}{6}]$. For 7.95 kbps frames, the search range is $[T_I - 10 \frac{2}{3}, T_I + 9 \frac{2}{3}]$. For 10.2 or 7.40 kbps frames, the search range is $[T_I - 5 \frac{2}{3}, T_I + 4 \frac{2}{3}]$. For all other bit rates, the search range is $[T_I - 5, T_I + 4]$.
2. The optimum integer pitch search minimizes the mean-square weighted error between the original and synthesized speech. This is achieved by maximizing the normalized cross-correlation given by

$$R(k) = \frac{\sum_{n=0}^{39} x(n)y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n)y_k(n)}}$$

where $x(n)$ is the target signal, and $y_k(n)$ is the past filtered excitation at delay k (past excitation convolved with the impulse response $h(n)$). The convolution $y_k(n)$ is computed for the first delay t_{\min} in the searched range. For other delays in the range $k = t_{\min} + 1, \dots, t_{\max}$, it is updated using the recursive relation

$$y_k(n) = y_{k-1}(n-1) + u(-k)h(n)$$

where $u(n)$, $n = -154, \dots, 39$, is the excitation history buffer. To simplify the search, the prediction residual is copied to $u(n)$, $n = 0, \dots, 39$, making it valid for all delays.

3. The fractional pitch search is performed by interpolating $R(k)$ and searching for its maximum. Fractional delay interpolation is achieved using an FIR filter, b_{24} , based on a hamming windowed *sinc* function truncated at ± 23 and padded with zeros at ± 24 , i.e.,

$$R_t(k) = \sum_{i=0}^3 R(k-i)b_{24}(t+6i) + \sum_{i=0}^3 R(k+1+i)b_{24}(6-t+6i)$$

where t corresponds to the fractional delay. If the bit rate is 12.2 kbps, then the delay is $-1/2$ to $1/2$ with a resolution of $1/6$; otherwise the fraction is $-2/3$ to $2/3$ with a resolution of $1/3$.

4. The adaptive codebook vector $v(n)$ is computed by interpolating the past excitation signal $u(n)$ at the integer delay k and fractional delay t , i.e.,

$$v(n) = \sum_{i=0}^9 u(n-k-i)b_{60}(t+6i) + \sum_{i=0}^9 u(n-k+1+i)b_{60}(6-t+6i)$$

The interpolation filter is based on a Hamming windowed *sinc* function truncated at ± 59 and padded with zeros at ± 60 .

5. For the first and the third subframes (only the first subframe during 4.75 and 5.15 kbps modes), the pitch lag bit allocation is rate-dependent. For 12.2 kbps frames, the pitch lag is encoded with 9 bits. For all other bit rates, the pitch lag is encoded with 8 bits. For the second and fourth subframes, pitch is encoded with 6 bits for the 12.2 and 7.95 kbps modes; 5 bits for 10.2 or 7.4 kbps modes, and 4 bits for all other modes.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcTarget</code> , <code>pSrcImpulseResponse</code> , <code>pSrcOpenLoopLag</code> , <code>pValResultPrevIntPitchLag</code> , <code>pSrcDstExcitation</code> , <code>pResultFracPitchLag</code> , <code>pResultAdptIndex</code> or <code>pDstAdptVector</code> pointer is NULL.

<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> is not in the range [0, 3].

AdaptiveCodebookDecode_GSMAMR

Decodes the adaptive codebook parameters.

Syntax

```
IppStatus ippAdaptiveCodebookDecode_GSMAMR_16s(Ipp16s valAdptIndex,
        Ipp16s* pValResultPrevIntPitchLag, Ipp16s* pValResultLtpLag,
        Ipp16s* pSrcDstExcitation, Ipp16s* pResultIntPitchLag,
        Ipp16s* pDstAdptVector, Ipp16s subFrame, Ipp16s bfi,
        Ipp16s inBackgroundNoise, Ipp16s voicedHangover, IppSpchBitRate mode);
```

Parameters

<i>valAdptIndex</i>	Adaptive codebook index.
<i>pValResultPrevIntPitchLag</i>	Pointer to the previous integer pitch lag. Used as an output argument also.
<i>pValResultLtpLag</i>	Pointer to the LTP-Lag value. Used as an output argument also.
<i>pSrcDstExcitation</i>	Pointer to the 194-element excitation vector. Elements 0 ~ 153 contain the past excitation, represented using Q15.0. Elements 154 ~ 193 are used as a buffer whenever the subframe length exceeds the pitch lag. On output, elements 154 - 193 are updated to contain the adaptive codebook vector.
<i>pResultIntPitchLag</i>	Pointer to the integer pitch.
<i>pDstAdptVector</i>	Pointer to the 40-sample adaptive codebook vector, represented using Q15.0.
<i>subFrame</i>	Subframe index.
<i>bfi</i>	Bad frame indicator. “0” signifies a good frame; any other value signifies a bad frame.

<i>inBackgroundNoise</i>	Flag set when the previous frame is considered to contain background noise and only shows minor energy level changes.
<i>voicedHangover</i>	Counter used to monitor the time since a frame was presumably voiced.
<i>mode</i>	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Description

The function `ippsAdaptiveCodebookDecode_GSMAMR` is declared in `ippsc.h` file. This function decodes the adaptive codebook parameters transmitted by the encoder, and then applies them to interpolate an adaptive codebook vector. If errors are detected on the received frame, previously received parameters are used to approximate the parameters of the current frame and the adaptive codebook vector interpolation procedure is carried with the approximated parameter set. Adaptive codebook vectors are decoded for every subframe. Details of this primitive are given next:

1. If no errors are detected on the current frame, integer and fractional pitch lags are extracted from the adaptive codebook indices.
2. If errors are detected, the integer pitch is recovered either from the previous integer pitch or the *LTP-Lag*, and the fractional pitch is set to zero. The *LTP-Lag* value is replaced by the integer pitch of the 4th subframe of the previous frame (12.2 Kbps mode) or slightly modified values based on the last correctly received value (all other modes).
3. The same adaptive codebook interpolation procedure described in section 13.4.3 is applied to obtain the adaptive codebook vector.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pValResultPrevIntPitchLag</i> , <i>pValResultLtpLag</i> , <i>pSrcDstExcitation</i> , <i>pResultIntPitchLag</i> , or <i>pDstAdptVector</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> is not in the range [0, 3].

AdaptiveCodebookGain_GSMAMR

Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.

Syntax

```
IppStatus ippsAdaptiveCodebookGain_GSMAMR_16s (const Ipp16s* pSrcAdptTarget,
    const Ipp16s* pSrcFltAdptVector, Ipp16s* pResultAdptGain);
IppStatus ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s (const Ipp16s*
    pSrcAdptTarget, const Ipp16s* pSrcFltAdptVector, Ipp16s* pResultAdptGain,
    Ipp16s* pResultAdptGainCoeffs);
```

Parameters

<i>pSrcAdptTarget</i>	Pointer to the adaptive target signal vector [40].
<i>pSrcFltAdptVector</i>	Pointer to the filtered adaptive-codebook vector [40], in Q12.
<i>pResultAdptGain</i>	Pointer to the output adaptive-codebook gain g_p , in Q14.
<i>pResultAdptGainCoeffs</i>	Pointer to the output gain coefficients vector [4] , in Q15.

Description

The functions `ippsAdaptiveCodebookGain_GSMAMR` and `ippsAdaptiveCodebookGainCoeffs_GSMAMR` are declared in `ippsc.h` file.

ippsAdaptiveCodebookGain_GSMAMR_16s. This function calculates the adaptive-codebook gain g_p as given by:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)}$$

bounded by $0 \leq g_p \leq 1.2$, in Q14,

where x is the adaptive target signal vector, and y is the filtered adaptive-codebook vector.

See also the `ippsAdaptiveCodebookGain_G729` function.

ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s. This function calculates the adaptive-codebook gain g_p in the same way as the `ippsAdaptiveCodebookGain_GSMAMR_16s` function does, and additionally returns the gain in a different representation given by the following formula:

$$g_p = \frac{c_{xy} 2^{\exp_{xy}}}{c_{yy} 2^{\exp_{yy}}}, 1/2 \leq |c_{xy}|, |c_{yy}| < 1, Q15$$

The mantissas c_{xy} , c_{yy} and exponents \exp_{xy} , \exp_{yy} for both the normalized denominator and normalized numerator are returned in the `pResultAdptGainCoeffs` vector:

`pResultAdptGainCoeffs[0] = cyy,`

`pResultAdptGainCoeffs[1] = expyy,`

`pResultAdptGainCoeffs[2] = cxy,`

`pResultAdptGainCoeffs[3] = expxy`

If $c_{xy} < 4$, then zero gain is returned.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the `pSrcAdptTarget`, `pSrcFltAdptVector`, `pResultAdptGain` or `pResultAdptGainCoeffs` pointer is NULL.

Fixed Codebook Search

This section describes primitives that are concerned with the fixed codebook, including primitives that perform the following functions:

- Fixed (algebraic) codebook search
- Fixed codebook vector decode

AlgebraicCodebookSearch_GSMAMR

Searches the algebraic codebook.

Syntax

```
IppStatus ippsAlgebraicCodebookSearch_GSMAMR_16s(Ipp16s valIntPitchLag,
    Ipp16s valBoundQAdptGain, const Ipp16s* pSrcFixedTarget,
    const Ipp16s* pSrcLtpResidual, Ipp16s* pSrcDstImpulseResponse,
    Ipp16s* pDstFixedVector, Ipp16s* pDstFltFixedVector,
    Ipp16s* pDstEncPosSign, Ipp16s subFrame, IppSpchBitRate mode);

IppStatus ippsAlgebraicCodebookSearchEX_GSMAMR_16s(Ipp16s valIntPitchLag,
    Ipp16s valBoundQAdptGain, const Ipp16s* pSrcFixedTarget,
    const Ipp16s* pSrcLtpResidual, Ipp16s* pSrcDstImpulseResponse,
    Ipp16s* pDstFixedVector, Ipp16s* pDstFltFixedVector, Ipp16s*
    pDstEncPosSign, Ipp16s subFrame, IppSpchBitRate mode, Ipp32s* pBuffer);
```

Parameters

<i>valIntPitchLag</i>	The nearest integer pitch lag T to the closed-loop fractional pitch lag of this subframe, which is computed by closed-loop pitch search routine.
<i>valBoundQAdptGain</i>	Bounded quantized adaptive codebook gain, which is denoted by the parameter β of the filter $F_E(z)$ in the ETSI GSM 06.90 standards document. For MR122 mode, this value is the bounded quantized pitch gain of current subframe. While for other modes, it is the bounded quantized pitch gain of previous subframe. This value is represented using Q1.14 format.
<i>pSrcFixedTarget</i>	Pointer to the 40-element fixed target signal vector $x_2(n)$, which is used to search the fixed codebook vector, represented using Q15.0. This should be aligned on an 8-byte boundary.
<i>pSrcLtpResidual</i>	Pointer to the 40-element long-term prediction residual signal vector $res_{LTP}(n)$, represented using Q15.0.
<i>pSrcDstImpulseResponse</i>	Pointer to the 40-element weighted synthesis filter impulse response vector, represented using Q3.12. This should be aligned on an 8-byte boundary. On output, points to the

	updated 40-element impulse response vector, which is obtained by filtering original impulse response $h(n)$ through the pre-filter $F_E(z)$. It is represented using Q3.12.
<i>pDstFixedVector</i>	Pointer to the 40-element fixed codebook vector $c(n)$, represented using Q2.13.
<i>pDstFltFixedVector</i>	Pointer to the 40-element filtered fixed codebook vector $z(n)$, which is obtained by convolving the impulse response with the fixed codebook vector, represented using Q2.13.
<i>pDstEncPosSign</i>	Pointer to the ten-element buffer that contains the encoded positions and signs of optimal pulses. For 12.2 kbps mode, 10 short words are used to store the result of this encoding. For the 10.2 kbps mode, 7 short words are used. For all other modes, only 2 short words are used.
<i>subFrame</i>	Subframe index, which ranges from 0 to 3.
<i>mode</i>	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.
<i>pBuffer</i>	Pointer to internal working buffer, of length 1K.

Description

The functions `ippsAlgebraicCodebookSearch_GSMAMR` and `ippsAlgebraicCodebookSearchEX_GSMAMR` are declared in `ippsc.h` file. These functions search the algebraic codebook by minimizing the mean square error between the weighted input speech and the weighted synthesized speech. After the fixed codebook vector has been obtained, it is filtered through the weighted synthesis filter to obtain a fixed codebook vector. The positions and signs of the optimal pulses are encoded respectively according to the GSM06.90 specification. Algebraic codebook search is applied on each subframe.

These two functions work identically with the following exception: `ippsAlgebraicCodebookSearchEX_GSMAMR` uses an internal working buffer pointed by *pBuffer* allocated by user, but `ippsAlgebraicCodebookSearch_GSMAMR` allocates this internal working buffer in stack.

Details of the algebraic search are given next.

1. Prior to fixed codebook search, if the integer part of the closed-loop pitch is less than the subframe length, filter the impulse response $h(n)$ through the pre-filter $F_E(z)$, i.e.,

$$h(n) = h(n) - \beta h(n-T) \quad , \quad n = T, \dots, 39$$
where β is the bounded and quantized adaptive gain, and T is the integer part of the closed pitch.

2. Compute backward filtered target vector \mathbf{d} by

$$d(n) = \sum_{i=n}^{39} x_2(i) \times h(i-n) \quad , \quad 0 \leq n \leq 39$$

where $x_2(n)$ is the fixed target signal used for fixed codebook search.

3. For 12.2 Kbps and 10.2 Kbps modes, the signal $b(n)$ which is used for presetting the pulse amplitudes is computed:

$$b(n) = \frac{res_{LTP}(n)}{\sqrt{\sum_{i=0}^{39} res_{LTP}(i) res_{LTP}(i)}} + \frac{d(n)}{\sqrt{\sum_{i=0}^{39} d(i) d(i)}} \quad , \quad n = 0, \dots, 39$$

For other modes, $b(n)$ is equal to the signal $d(n)$.

4. Calculate the symmetric Toeplitz matrix Φ for each mode, the element of it is computed by

$$\Phi(i, j) = \sum_{n=j}^{39} h(n-i) \times h(n-j) \quad , \quad i \leq j \quad , \quad 0 \leq i \leq 39$$

5. Search the optimal pulse positions for each mode. An efficient non-exhaustive analysis-by-synthesis search technique is used for 12.2Kbps, 10.2Kbps, 7.95Kbps, 7.4Kbps, 6.70 Kbps modes. An exhaustive analysis-by-synthesis search technique is used for other modes. GSM 06.90 (clause 5.7) contains complete details.
6. Construct the fixed codebook vector $c(n)$ according to optimal pulse positions, then convolve it with $h(n)$ to get filtered fixed codebook vector $z(n)$

$$z(n) = \sum_{i=0}^{39} c(i) \times h(n-i) \quad , \quad 0 \leq n \leq 39$$

Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

FixedCodebookDecode_GSMAMR

Decodes the fixed codebook vector.

Syntax

```
IppStatus ippFixedCodebookDecode_GSMAMR_16s(const Ipp16s* pSrcFixedIndex,
      Ipp16s* pDstFixedVector, Ipp16s subFrame, IppSpchBitRate mode);
```

Parameters

<i>pSrcFixedIndex</i>	Pointer to the fixed codebook index vector. If the mode is 12.2 kbps, the vector length is 10; if the mode is 10.2 kbps, the vector length is 7; otherwise the vector length is 2.
<i>pDstFixedVector</i>	Pointer to the 40-element fixed codebook vector.
<i>subFrame</i>	Subframe index.
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

Description

The function `ippFixedCodebookDecode_GSMAMR` is declared in `ippsc.h` file. This function decodes the fixed codebook vector from the received fixed codebook index.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when the <i>pSrcFixedIndex</i> or <i>pDstFixedVector</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> is not in the range <code>[0, 3]</code> .

Discontinuous Transmission (DTX)

This section describes primitives that are concerned with discontinuous transmission (DTX), including primitives that perform the following functions:

- Signal pre-emphasis prior to VAD option 2
- VAD Decision Function for Option 1
- VAD Decision Function for Option 2
- Parameter Extraction for the SID frame
- DTX Handler
- DTX Buffering

Each of these primitives is described next.

Preemphasize_GSMAMR

Computes pre-emphasis of an input signal in VAD option 2.

Syntax

```
IppStatus ippsPreemphasize_GSMAMR_16s (Ipp16s gamma, const Ipp16s* pSrc,
    Ipp16s* pDst, int len, Ipp16s* pMem);
```

Parameters

<i>gamma</i>	The filter coefficient, in Q15.
<i>pSrc</i>	Pointer to the source vector, in Q0.
<i>pDst</i>	Pointer to the destination vector, in Q0.
<i>len</i>	Number of elements in the source and destination vectors.
<i>pMem</i>	Pointer to the filter memory [1].

Description

The function `ippsPreemphasize_GSMAMR` is declared in the `ippsc.h` file. This function computes pre-emphasis of the input signal prior to frequency domain conversion in Voice Activity Detector option 2. The function `ippsPreemphasize_GSMAMR` performs the same operation as the [ippsPreemphasize_G729A](#) function does, but has a slightly different order to deliver accuracy required by the GSM-AMR transcoding standard.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pMem</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

VAD1_GSMAMR

Implements the VAD functionality corresponding to VAD option 1.

Syntax

```
IppStatus ippsVAD1_GSMAMR_16s(const Ipp16s pSrcSpch, IppGSMAMRVad1State*
    pValResultVad1State, Ipp16s* pResultVadFlag, Ipp16s maxHpCorr,
    Ipp16s toneFlag);
```

Parameters

<code>pSrcSpch</code>	Pointer to the input speech signal, of length 160, in Q0.
<code>pValResultVad1State</code>	On input, pointer to the VAD Option 1 history variables. On output, points to the updated VAD Option 1 history variables. The structure <code>IppGSMAMRVad1State</code> is defined below.
<code>pResultVadFlag</code>	Pointer to the VAD flag of this frame. If it is set to “1”, it indicates the presence of signals that should be transmitted. If set to “0”, there is no signals in this frame needed to be transmitted.

<i>maxHpCorr</i>	<i>best_corr_hp</i> value of previous frame, which is the maximum normalized value of the high pass filtered correlation. This value is the output of the open-loop pitch search function.
<i>toneFlag</i>	Tone flag, which indicates the presence of information tones or signals containing very strong periodic component. This value is the output of the open-loop pitch search function.

Description

The function `ippsVAD1_GSMAMR` is declared in `ippsc.h` file. This function implements the VAD functionality corresponding to VAD option 1 of *ETSI GSM 06.94*. It is used to indicate whether each 20ms frame contains signals that should be transmitted - for example, speech, music or information tones. The structure `IppGSMAMRVad1State` contains the history variables of VAD Option 1. These variables are initialized before the beginning of the encoder, and can be only updated by this function. Refer to *ETSI GSM 06.94* VAD Option 1 specification for details of the implementation.

Structure definition of IppGSMAMRVad1State:

typedef struct{	Description
<code>Ipp16s pPrevSignalLevel[9];</code>	Signal level vector of <i>level[n]</i> previous frame.
<code>Ipp16s pPrevSignalSublevel[9];</code>	Intermediate signal sublevel vector of previous frame.
<code>Ipp16s pPrevAverageLevel[9];</code>	Average signal level vector <i>ave_level[n]</i> of previous frame.
<code>Ipp16s pBkgNoiseEstimate[9];</code>	Background noise estimate vector <i>back_est[n]</i> of previous frame.
<code>Ipp16s pFifthFltState[6];</code>	The history state of the three 5 th order filters of filter bank.
<code>Ipp16s pThirdFltState[5];</code>	The history state of the five 3 rd order filters of filter bank.
<code>Ipp16s burstCount;</code>	Burst counter <i>burst_count</i> which counts length of a speech burst, used by VAD hangover addition.
<code>Ipp16s hangCount;</code>	Hang counter <i>hang_counter</i> which is used by VAD hangover addition.
<code>Ipp16s statCount;</code>	Stationary counter variable <i>stat_count</i> which is used in background noise estimation.
<code>Ipp16s vadReg;</code>	Value that indicates intermediate VAD decision.
<code>Ipp16s complexHigh;</code>	<i>complex_high</i> value which is used as intermediate complex signal decision.
<code>Ipp16s complexLow;</code>	<i>complex_low</i> value which is used as intermediate complex signal decision.
<code>Ipp16s complexHangTimer;</code>	<i>complex_hang_timer</i> which is used as hangover initiator by Complex Activity Estimation.
<code>Ipp16s complexHangCount;</code>	<i>complex_hang_count</i> which is used as hangover counter by VAD hangover addition.
<code>Ipp16s complexWarning;</code>	<i>complex_warning</i> flag.
<code>Ipp16s corrHp;</code>	The high-pass filtered value of <i>best_corr_hp</i> .
<code>Ipp16s pitchFlag;</code>	Pitch flag which indicates the presence of vowel sounds and other periodic signals.
<code>}IppGSMAMRVad1State.</code>	

Note. VAD option 1 history variables initialization:

Whenever the Encoder is reset, all elements in `pPrevSignalLevel`, `pPrevSignalSublevel`, `pPrevAverageLevel` vector should be set as 150, `corrHp` should be set as 13106. While all other variables should be initialized as 0.

For the detail usage of these history variables, please refer to *ETSI GSM 06.94*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.

VAD2_GSMAMR

Implements the VAD functionality corresponding to VAD option 2.

Syntax

```
IppStatus ippSVAD2_GSMAMR_16s(const Ipp16s* pSrcSpch, IppGSMAMRVad2State*
    pValResultVad2State, Ipp16s* pResultVadFlag, Ipp16s ltpFlag);
```

Parameters

<code>pSrcSpch</code>	Pointer to the input speech frame, of length 160, in Q0.
<code>pValResultVad2State</code>	On input, pointer to the VAD Option 2 history variables. On output, points to the updated VAD Option 2 history variables. The structure <code>IppGSMAMRVad2State</code> is defined below.
<code>pResultVadFlag</code>	Pointer to the Boolean flag <code>VAD_flag</code> . If it is set to “1”, it indicates the presence of signals that should be transmitted. If set to “0”, there is no signals in this frame needed to be transmitted.
<code>ltpFlag</code>	<i>LTP_flag</i> value in <i>GSM 06.94</i> equation (4.24), which is generated by the comparison of the long-term prediction to a constant threshold <i>LTP_THLD</i> .

Description

The function `ippSVAD2_GSMAMR` is declared in `ippsc.h` file. This function implements the VAD functionality corresponding to VAD option 2 of *ETSI GSM 06.94*. It is used to indicate whether each 20ms frame contains signals that should be transmitted. For example, speech, music, or information tones. The structure `IppGSMAMRVad2State` contains the history variables of VAD Option 2.

Please refer to *ETSI GSM 06.94 VAD Option 2* specification for details.

Structure definition of `IppGSMAMRVad2State`:

typedef struct{	Description
<code>Ipp32s pEngyEstimate[16];</code>	Channel energy estimates vector \mathbf{E}_{ch} of current half-frame, which is calculated during the previous half-frame according to the equation <i>ETSI GSM 06.94 (4.4)</i> .
<code>Ipp32s pNoiseEstimate[16];</code>	Channel noise estimate vector \mathbf{E}_n of current half-frame, which is calculated during the previous half-frame according to the equation <i>ETSI GSM 06.94 (4.26)</i> .
<code>Ipp16s pLongTermEngyDb[16];</code>	Channel average long-term spectral estimate vector \mathbf{E}_{dB} , which is calculated during the previous half-frame according to the equation <i>ETSI GSM 06.94 (4.20)</i> .
<code>Ipp16s preEmphasisFactor;</code>	Pre-emphasis factor ζ_p , which is used to pre-emphasize the input speech signal according to the equation <i>ETSI GSM 06.94 (4.1)</i> .
<code>Ipp16s updateCount;</code>	<i>update_cnt</i> value used in background noise update decision logic.
<code>Ipp16s lastUpdateCount;</code>	<i>last_update_cnt</i> value used in background noise update decision logic.
<code>Ipp16s hysterCount;</code>	<i>hyster_cnt</i> value used in background noise update decision logic.
<code>Ipp16s prevNormShift;</code>	Shifted bits of previous half-frame input speech when normalized to obtain high precision and avoid overflow when doing FFT transformation.
<code>Ipp16s shiftState;</code>	Shift state flag which indicates whether previous half-frame has been shifted or not.
<code>Ipp16s forcedUpdateFlag;</code>	<i>fupdate_flag</i> value which is the result of forced update logic of background noise update decision.
<code>Ipp16s ltpSnr;</code>	Long-term peak signal-noise ratio SNR_p of previous half-frame, which is used to calibrate the responsiveness of VAD decision.
<code>Ipp16s variabFactor;</code>	Variability factor ψ of previous half-frame, which indicates the variability of the background noise estimate and is updated according to the equation <i>ETSI GSM 06.94 (4.13)</i> .
<code>Ipp16s negSnrBias;</code>	Negative SNR sensitivity Bias factor μ of previous half-frame.
<code>Ipp16s burstCount;</code>	Burst counter $b(m)$ used in the 10 ms half-frames's VAD Decision.
<code>Ipp16s hangOverCount;</code>	Hangover counter $h(m)$ used in the 10ms half-frame's VAD Decision.
<code>Ipp32s frameCount;</code>	Half-frame counter.
<code>}IppGmrVad2State;</code>	

Note. All elements in the structure should be initialized to zero whenever encoder is reset.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.

EncDTXSID_GSMAMR

Extracts parameters for the SID frame.

Syntax

```
IppStatus ippEncDTXSID_GSMAMR_16s(const Ipp16s* pSrcLspBuffer, const
    Ipp16s* pSrcLogEnergyBuffer, Ipp16s* pValResultLogEnergyIndex,
    Ipp16s* pValResultDtxLsfRefIndex, Ipp16s* pSrcDstQLsfIndex,
    Ipp16s* pSrcDstPredQErr, Ipp16s* pSrcDstPredQErrMR122, Ipp16s sidFlag);
```

Parameters

<i>pSrcLspBuffer</i>	Pointer to the LSP coefficients of eight consecutive frames marked with VAD = 0, in the length of 80, in Q0.15.
<i>pSrcLogEnergyBuffer</i>	Pointer to the log energy coefficients of eight consecutive frames marked with unvoiced, in the length of 8, in Q5.10.
<i>pValResultLogEnergyIndex</i>	Pointer to log energy index of last frame, in Q2.13. On output, points to the log energy index of current frame, in Q2.13.
<i>pValResultDtxLsfRefIndex</i>	Pointer to the LSF quantization reference index of last frame. On output, points to the LSF quantization reference index of current DTX frame.
<i>pSrcDstQLsfIndex</i>	Pointer to the LSF residual quantization indices of last frame, in the length of 3. On output, points to the LSF residual quantization indices of current frame, in the length of 3.

<i>pSrcDstPredQErr</i>	Pointer to the fixed gain prediction error of four previous subframes for non-12.2 Kbps modes, in the length of 4, in Q5.10. On output, points to the updated fixed gain prediction error for non 12.2 Kbps modes, in the length of 4, in Q5.10.
<i>pSrcDstPredQErrMR122</i>	Pointer to the fixed gain prediction error of four previous subframes for 12.2 Kbps, in the length of 4, in Q5.10. On output, points to the updated fixed gain prediction error for 12.2 Kbps mode, in the length of 4, in Q5.10.
<i>sidFlag</i>	The SID flag of the current frame. If it is set to 1, the current frame is a SID frame, and the function will extract the LSF and energy parameters. If it is set to 0, the LSF and energy parameters will copy from previous frame.

Description

The function `ippsEncDTXSID_GSMAMR` is declared in `ippsc.h` file. This function is called only when the current frame is a DTX frame. It extracts the needed parameters for the SID frame (that is, the LSF quantization parameter and the energy index parameter). If the SID flag is off, no operation is needed, and all the parameters are copied from last frame. If the SID flag is on, the functionality is as below:

1. Compute the log energy index

$$EnLogIndex = \frac{1}{8} \sum_{n=0}^7 EnLog(t-n) ,$$

where $EnLog(t)$ is the log energy of the current frame in log2 scale.

2. Update the fixed gain prediction error for 12.2 Kbps mode and other modes respectively with the log energy index:

$$PredErr(i) = EnLogIndex$$

$$PredErrMR122(i) = (EnLogIndex) / (20 \times \log_{10}(2))$$

$$i = 0, \dots, 3$$

3. Compute the average LSP coefficients of the current frame and past seven frames:

$$Lsp_{mean}(i) = \frac{1}{8} \sum_{n=0}^7 Lsp(i-n)$$

4. Quantize the average LSP coefficients:

- a. Convert the average LSP to LSF, then reorder LSF and multiply LSF with weighting coefficients.
- b. Find the LSF reference vector:

$$Lsfref_Index = \min_{index} \left(\sum_{n=0}^9 (Lsf(n) - Lsf_ref_{index}(n))^2 \mid index = 0, \dots, 8 \right)$$

- c. Get the LSF residual:

$$Lsf_residual(i) = Lsf(i) - Lsf_ref_{index}(i), \quad i = 0, \dots, 9$$

- d. Quantize the LSF residual, using split band vector quantization method.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.

EncDTXHandler_GSMAMR

Determines the SID flag of current frame.

Syntax

```
IppStatus ippEncDTXHandler_GSMAMR_16s(Ipp16s* pValResultHangOverCount,
    Ipp16s* pValResultDtxElapsedCount, Ipp16s* pValResultUsedMode,
    Ipp16s* pResultSidFlag, Ipp16s vadFlag);
```

Parameters

<code>pValResultHangOverCount</code>	Pointer to the DTX hangover count. When initialized or reset, it is set to 0. On output, points to the updated DTX hangover count.
<code>pValResultDtxElapsedCount</code>	Pointer to elapsed frame count since last non-DTX frame. When initialized or reset, it is set 0. On output, points to the updated elapsed frame count since last non-DTX frame.

<i>pValResultUsedMode</i>	Pointer to the transmission mode. At the input stage, the mode is one of the bit rate modes ranging from 4.75 Kbps to 12.2 Kbps. At the output stage, this value is either unchanged or set to the DTX frame mode.
<i>pResultSidFlag</i>	Pointer to the output SID flag, “1” indicates a SID frame, and “0” indicates a non-SID frame.
<i>vadFlag</i>	This is the VAD flag of the current frame, if it is set 1, the current frame is marked with voiced, and if it is set to 0, it is marked with unvoiced.

Description

The function `ippEncDTXHandler_GSMAMR` is declared in `ippsc.h` file. This function determines the SID flag of current frame, and it determines whether the current frame should use DTX encoding.

1. Update the elapsed frame count since last SID frame: $DTX_ElapsedCount = DTX_ElapsedCount + 1$ (Bounded to 0~0x7fff).
2. If the VAD flag of current frame is 1 (voiced frame), the DTX hangover count is set to 7, and this function ends.
3. If the VAD flag of current frame is 0 (unvoiced frame), and the DTX hangover count is 0, the transmission mode of this frame is set to DTX frame mode, and the elapsed frame count since last DTX frame is set to 0, the SID flag is set to 1.
4. If the VAD flag of current frame is 0 (unvoiced frame), but the DTX hangover count is not 0, then decrease DTX hangover count by 1, and if $DTX_HangOver_Count + DTX_ElapsedCount < 30$ The SID flag is set to 0, and the transmission mode is set to “MRDTX”.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.

EncDTXBuffer_GSMAMR , DecDTXBuffer_GSMAMR

Buffer the LSP (or LSF) coefficients and previous log energy coefficients.

Syntax

```
IppStatus ippsEncDTXBuffer_GSMAMR_16s(const Ipp16s* pSrcSpch,
    const Ipp16s* pSrcLsp, Ipp16s* pValResultUpdateIndex,
    Ipp16s* pSrcDstLspBuffer, Ipp16s* pSrcDstLogEnergyBuffer);

IppStatus ippsDecDTXBuffer_GSMAMR_16s(const Ipp16s* pSrcSpch,
    const Ipp16s* pSrcLsf, Ipp16s* pValResultUpdateIndex,
    Ipp16s* pSrcDstLsfBuffer, Ipp16s* pSrcDstLogEnergyBuffer);
```

Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal, in the length of 160, in Q15.0
<i>pSrcLsp</i>	Pointer to the LSP for this frame, in the length of 10, in Q0.15.
<i>pSrcLsf</i>	Pointer to the LSF coefficients of the current frame, in the length of 10, in Q0.15.
<i>pValResultUpdateIndex</i>	Pointer to the previous memory update index. On output, points to the current memory update index. It is circularly increased between 0 and 7.
<i>pSrcDstLspBuffer</i>	Pointer to the LSP coefficients of eight previous frames, in the length of 80, in Q0.15. On output, points to the LSP coefficients of eight most recent frames (including current frame), in the length of 80, in Q0.15.
<i>pSrcDstLsfBuffer</i>	Pointer to the LSF coefficients of eight previous frames, in the length of 80, in Q0.15.
<i>pSrcDstLogEnergyBuffer</i>	Pointer to the logarithm energy coefficients of eight previous frames, in the length of 8, in Q5.10. On output, points to the log energy coefficients of eight most recent frames (including current frame), in the length of 8, in Q5.10.

Description

The functions `ippsEncDTXBuffer_GSMAMR` and `ippsDecDTXBuffer_GSMAMR` are declared in `ippsc.h` file. These functions buffer the LSP (or LSF) coefficients and previous log energy coefficients. These LSPs (or LSFs) and energy coefficients will be used for SID frame to extract necessary parameters. The memory update index indicates which part of the buffer will be updated, and it saves the cost for some memory copy. The log energy is computed as follows:

$$EnLog = \log_2 \left(\frac{1}{N} \sum_{n=0}^{N-1} s^2(n) \right),$$

where N is the frame length, and s is the input speech signal.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.

Post Processing

PostFilter_GSMAMR

Filters the synthesized speech.

Syntax

```
IppStatus ippsPostFilter_GSMAMR_16s(const Ipp16s* pSrcQLpc,
    const Ipp16s* pSrcSpch, Ipp16s* pValResultPrevResidual,
    Ipp16s* pValResultPrevScalingGain, Ipp16s* pSrcDstFormantFIRState,
    Ipp16s* pSrcDstFormantIIRState, Ipp16s* pDstFltSpch, IppSpchBitRate mode);
```

Parameters

<code>pSrcQLpc</code>	Pointer to the reconstructed LP coefficients, in the length of 44, in Q3.12.
<code>pSrcSpch</code>	Pointer to the start position of the input speech signal for current frame, in the length of 160, in Q15.0.

<i>pValResultPrevResidual</i>	<p>On entry, pointer to the last output of the FIR filter of the formant filter for previous subframe, in Q15.0. It is the input of the tilt compensation filter.</p> <p>On exit, points to the last output of the FIR filter of the formant filter for this subframe (in Q15.0) and is the output of the tilt compensation filter.</p> <p>This value is initialized to 0 and can only be updated by this function.</p>
<i>pValResultPrevScalingGain</i>	<p>Pointer to the scaling factor b of the last signal for the previous subframe, in Q3.12. On output, points to the scaling factor b of the last signal for this subframe, in Q3.12</p>
<i>pSrcDstFormantFIRState</i>	<p>Pointer to the state of the FIR part of the formant filter, in the length of 10, in Q15.0. On output, points to the updated state of the FIR part of the formant filter, in the length of 10, in Q15.0.</p>
<i>pSrcDstFormantIIRState</i>	<p>Pointer to the state of the IIR part of the formant filter, in the length of 10, in Q15.0. On output, points to the updated state of the IIR part of the formant filter, in the length of 10, in Q15.0.</p>
<i>pDstFltSpch</i>	<p>Pointer to the filtered speech, in the length of 160, in Q15.0.</p>
<i>mode</i>	<p>The source transmission mode, for this function, the value between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 is valid.</p>

Description

The function `ippsPostFilter_GSMAMR` is declared in `ippsc.h` file. This function filters the synthesized speech to enhance reconstruction quality. This primitive implements the following procedure:

1. Get the weighted LP coefficients of the formant postfilter:

$$H_F(z) = \frac{\hat{A}(z/\gamma_n)}{A(z/\gamma_d)}$$

For 12.2 and 10.2 Kbps modes,

$$\gamma_n = 0.7, \quad \gamma_d = 0.75$$

For other modes,

$$\gamma_n = 0.55, \quad \gamma_d = 0.7$$

Then filter the input speech with $HF(z) = \hat{A}(z/\gamma_n)$.

2. Compute the impulse response $h(n)$ of the formant postfilter, then get the first reflection coefficients:

$$k'_1 = \frac{r_h(1)}{r_h(0)}, \quad r_h(i) = \sum_{j=0}^{21-i} h(j) \times h(j+i)$$

Then compute the tilt factor: $\mu = \gamma_t k'_1$

For 12.2 and 10.2 Kbps mode,

$$\gamma_t = \begin{cases} 0.8 & k'_1 > 0, \\ 0 & \text{otherwise} \end{cases}$$

For other modes, $\gamma_t = 0.8$

Then filter the signal through the tilt compensation filter: $H_t(z) = 1 - \mu z^{-1}$

The output of the tilt compensation filter is filtered by:

$$HI(z) = \hat{A}(z/\gamma_d)$$

3. Compute the adaptive gain scaling factor:

$$\gamma_{SC} = \sqrt{\frac{\sum_{n=0}^{39} \hat{s}(n)}{\sum_{n=0}^{39} \hat{s}_F(n)}}$$

Here, $\hat{s}(n)$ is the synthesized speech, and $\hat{s}_f(n)$ is the post-filtered speech. The output speech is given by:

$$\hat{s}'(n) = \beta_{sc}(n)\hat{s}_f(n)$$

The scaling factor is updated sample-by-sample:

$$\beta_{sc}(n) = 0.9\beta_{sc}(n-1) + 0.1\gamma_{sc}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is <code>NULL</code> . r the input variable <code>mode</code> is out of range.
<code>ippStsRangeErr</code>	Indicates an error when the input variable <code>mode</code> is out of range.

AMR Wideband Related Functions

The Intel IPP functions described in this section if properly combined can be used to construct the Adaptive Multi-Rate Wideband (AMR WB) Speech Codec compliant to 3rd Generation Partnership Project (3GPP) specification TS 26.173: “*AMR Wideband Speech Codec; ANSI-C code*”. The description of AMR WB codec may be found in 3GPP TS 26.190: “*AMR Wideband Speech Codec; Transcoding functions*”. Also it is known as ITU-T G.722.2 AMR WB codec.

The primitives are primarily designed to implement the well-defined, computationally expensive core operations that comprise the codec portion of the AMR WB system. The AMR WB codec comprises an adaptive multi-rate algorithm intended for encoding 7 kHz bandwidth speech signals at the sampling rate of 16 000 samples per second, which results in bit rates for the encoded bit stream of 6.60, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s.

The list of Intel IPP AMR WB functions is given in the following table:

Table 9-5 Intel IPP AMR WB Related Functions

Function Base Name	Operation
ResidualFilter_AMRWB	Computes the LPC residual
LPCToISP_AMRWB	Performs LP to ISP coefficients conversion
ISPToLPC_AMRWB	Performs ISP to LP coefficients conversion.
ISPToISF_Norm_AMRWB	Performs ISP to ISF coefficients conversion
ISFTToISP_AMRWB	Performs ISF conversion to ISP

Table 9-5 Intel IPP AMR WB Related Functions (continued)

Function Base Name	Operation
<u>OpenLoopPitchSearch_AMRWB</u>	Extracts an open-loop pitch lag estimate from the weighted input speech
<u>HighPassFilterGetSize_AMRWB</u>	Returns the size of the high-pass filter state memory
<u>HighPassFilterInit_AMRWB</u>	Initializes the state memory of high-pass filter
<u>HighPassFilter_AMRWB</u>	Performs high-pass filtering
<u>Preemphasize_AMRWB</u>	Computes pre-emphasis of a speech signal
<u>Deemphasize_AMRWB</u>	Performs de-emphasis filtering
<u>SynthesisFilter_AMRWB</u>	Reconstructs the speech signal from LP coefficients and residuals
<u>VADGetSize_AMRWB</u>	Returns the size of the VAD module state memory
<u>VADInit_AMRWB</u>	Initializes the VAD module state memory
<u>VAD_AMRWB</u>	Performs VAD in AMR WB encoder
<u>AlgebraicCodebookSearch_AMRWB</u>	Performs the fixed (algebraic) codebook search
<u>AlgebraicCodebookDecode_AMRWB</u>	Decodes the fixed (algebraic) codebook indexes
<u>AdaptiveCodebookGainCoeff_AMRWB</u>	Computes the adaptive codebook gain
<u>AdaptiveCodebookSearch_AMRWB</u>	Performs the adaptive codebook search
<u>AdaptiveCodebookDecodeGetSize_AMRWB</u>	Queries the memory length of the adaptive codebook decode module
<u>AdaptiveCodebookDecodeInit_AMRWB</u>	Initializes the adaptive codebook decode module memory
<u>AdaptiveCodebookDecodeUpdate_AMRWB</u>	Updates the adaptive codebook decode module memory
<u>AdaptiveCodebookDecode_AMRWB</u>	Performs the adaptive codebook decoding
<u>ISFQuant_AMRWB</u>	Quantizes the ISF
<u>ISFQuantDecode_AMRWB</u>	Decodes quantized ISFs from the received codebook index
<u>ISFQuantDTX_AMRWB</u>	Quantizes the ISF coefficient vector in case of DTX mode
<u>ISFQuantDecodeDTX_AMRWB</u>	Decodes quantized ISFs in case of DTX mode
<u>GainQuant_AMRWB</u>	Quantizes the adaptive codebook gains
<u>DecodeGain_AMRWB</u>	Decodes adaptive and fixed-codebook gains
<u>EncDTXBuffer_AMRWB</u>	Buffers the ISP coefficients and previous logarithm energy coefficients
<u>DecDTXBuffer_AMRWB</u>	Buffers the ISF coefficients and previous logarithm energy coefficients.

The use of these functions is demonstrated in the Intel IPP *GSM/AMR Wideband/G.722.2 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

ResidualFilter_AMRWB

Computes the LPC residual

Syntax

```
IppStatus ippsResidualFilter_AMRWB_16s_Sfs (const Ipp16s* pSrcLpc, int order,
      const Ipp16s* pSrcSpeech, Ipp16s* pDstResidualSignal, int len, int
      scaleFactor);
```

Parameters

<i>pSrcLpc</i>	Pointer to the input LPC.
<i>order</i>	Length of the LPC vector.
<i>pSrcSpeech</i>	Pointer to the input vector [- <i>order</i> ,...,-1,0,..., <i>len</i> -1].
<i>pDstResidualSignal</i>	Pointer to the output vector of length [<i>len</i>].
<i>len</i>	Length of the vectors.
<i>scaleFactor</i>	Scale factor value.

Description

This function is declared in `ippsc.h` file. The functionality is the same as for the function [ippsResidualFilter_G729E_16s](#). The only difference is that the function `ippsResidualFilter_AMRWB` scales the result according to *scaleFactor* value.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcLpc</i> , <i>pSrcSpeech</i> or <i>pDstResidualSignal</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>order</i> or <i>len</i> is less than or equal to zero, or when <i>order</i> is greater than <i>len</i> .

`ippStsScaleRangeErr` Indicates an error when *scaleFactor* is negative or *scaleFactor* is greater than 15.

LPC Analysis Functions

LPCToISP_AMRWB

Performs LP to ISP coefficients conversion.

Syntax

```
IppStatus ippSLPCToISP_AMRWB_16s( const Ipp16s* pSrcLpc, Ipp16s* pDstIsp,
    const Ipp16s* pSrcPrevIsp);
```

Parameters

<i>pSrcLpc</i>	Pointer to the input predictor coefficients.
<i>pDstIsp</i>	Pointer to the output immittance spectral pairs.
<i>pSrcPrevIsp</i>	Pointer to the input previous immittance spectral pairs.

Description

This function is declared in `ippsc.h` file. The function performs the following steps:

1. Calculates the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the following recursive relations:

$$i = 0, 1, \dots, 7,$$

$$f_1(i) = a_i + a_{m-i}$$

$$f_2(i) = a_i - a_{m-i}$$

$$f_1(8) = 2a_8$$

$$\text{where } f_2(-2) = f_2(-1) = 1.0.$$

2. Uses Chebyshev polynomials to evaluate $F_1(z)$ and $F_2(z)$. The Chebyshev polynomials are given by:

$$C_1(\omega) = \sum_{i=0}^7 f_1(i) \cos((8-i)\omega) + f_1(8)/2$$

$$C_2(\omega) = \sum_{i=0}^6 f_2(i) \cos((8-i)\omega) + f_2(7)/2$$

$F_1(z)$ and $F_2(z)$ polynomials are evaluated at 100 points in equally spaced intervals between 0 and π and are checked for sign changes. A sign change indicates the existence of a root in corresponding interval, which is divided then four times to track the root.

3. If all 16 roots needed to determine ISP coefficients are not found, the function returns the previous set of ISP coefficients instead.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcLpc</i> , <i>pDstIsp</i> or <i>pSrcPrevIsp</i> pointer is NULL.

ISPToLPC_AMRWB

Performs ISP to LP coefficients conversion.

Syntax

```
IppStatus ippISPToLPC_AMRWB_16s(const Ipp16s* pSrcIsp, Ipp16s* pDstLpc, int len);
```

Parameters

<i>pSrcIsp</i>	Pointer to the input immittance spectral pairs.
<i>pDstLpc</i>	Pointer to the output predictor coefficients.
<i>len</i>	Number of ISP and LPC.

Description

This function is declared in `ippsc.h` file. The function performs the following steps:

1. Calculates the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursive relations from i equal to 2 to $len/2$ by formulae

$$f_1(i) = -2q_{2i-2}f_1(i-1) + 2f_1(i-2)$$

for $j=i-1$ down to 2

$$f_1(j) = f_1(j) - 2q_{2i-2}f_1(j-1) + 2f_1(j-2)$$

end

$$f_1(1) = f_1(1) - 2q_{2i-2}$$

with initial values $f_1(0) = 1$ and $f_1(1) = -2q_0$.

The coefficients $f_2(i)$ are computed similarly by replacing q_{2i-2} by q_{2i-1} and $len/2$ by $len/2-1$, and with initial conditions $f_2(0) = 1$ and $f_2(1) = -2q_1$.

2. Once the coefficients $f_1(z)$ and $f_2(z)$ are found, $F_2(z)$ is multiplied by $1-z^{-2}$ to obtain $F_2'(z)$, that is

$$f_2'(i) = f_2(i) - f_2(i-2) , \quad i = 2, \dots, len/2 - 1$$

$$f_1'(i) = f_1(i) , \quad i = 0, \dots, len/2$$

Then $F_1'(z)$ and $F_2'(z)$ are multiplied by $1+q_{len-1}$ and $1-q_{len-1}$, respectively. That is

$$f_2'(i) = (1 - q_{m-1})f_2'(i) , \quad i = 0, \dots, len/2 - 1$$

$$f_1'(i) = (1 + q_{m-1})f_1'(i) , \quad i = 0, \dots, len/2$$

3. Finally, the function computes the LP coefficients from $f_1'(i)$ and $f_2'(i)$ as:

$$a_i = \begin{cases} 0.5f_1'(i) + 0.5f_2'(i) , & i = 1, \dots, len/2 - 1 \\ 0.5f_1'(i) - 0.5f_2'(i) , & i = len/2 + 1, \dots, len-1 \\ 0.5f_1'(len/2) , & i = len/2 \\ q_{len-1} , & i = len \end{cases}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pSrcIsp</i> or <i>pDstLpc</i> pointer is NULL.
<code>IppStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero, or when <i>len</i> is greater than 20.

ISPToISF_Norm_AMRWB

Performs ISP to ISF coefficients conversion.

Syntax

```
IppStatus ippSISPToISF_Norm_AMRWB_16s(const Ipp16s* pSrcIsp, Ipp16s* pDstIsf,
int len);
```

Parameters

<i>pSrcIsp</i>	Pointer to the ISP input vector of values scaled in range [-1:1].
<i>pDstIsf</i>	Pointer to the ISF output vector of values scaled in range [0:0.5].
<i>len</i>	Number of ISP and ISF.

Description

This function is declared in `ippsc.h` file. The function `ippSISPToISF_Norm_AMRWB_16s` converts the ISP coefficients to ISF coefficients as follows:

$$pDstIsf[i] = \begin{cases} \frac{f_s}{2\pi} \arccos(pSrcIsp[i]), & i = 0, \dots, len - 1 \\ \frac{f_s}{4\pi} \arccos(pSrcIsp[i]) & i = len \end{cases}$$

Here $f_s = 12800$ is the sampling frequency. The scale factor is chosen such that the first ISF coefficient is normalized to the interval [0:0.5] by multiplying to $1/\pi$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcIsf</i> or <i>pDstIsf</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

ISFToISP_AMRWB

Performs ISF conversion to ISP.

Syntax

```
IppStatus ippISFToISP_AMRWB_16s(const Ipp16s* pSrcIsf, Ipp16s* pDstIsf, int
    len);
```

Parameters

<i>pSrcIsf</i>	Pointer to the ISF input vector of values scaled in range [0:0.5].
<i>pDstIsf</i>	Pointer to the ISP output vector of values scaled in range [-1:1].
<i>len</i>	Number of ISP and ISF.

Description

This function is declared in `ippsc.h` file. It converts ISF to the ISP coefficients as given by:

$$pDstIsf[i] = \cos(pSrcIsf[i]), \quad i = 0, \dots, len$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcIsf</i> or <i>pDstIsf</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

Open-loop Pitch Analysis Functions

OpenLoopPitchSearch_AMRWB

Extracts an open-loop pitch lag estimate from the weighted input speech.

Syntax

```
IppStatus ippsOpenLoopPitchSearch_AMRWB_16s(const Ipp16s* pSrcWgtSpch,
      const Ipp16s* pSrcFltWgtSpch, Ipp16s* pPrevMidPitchLag, Ipp16s*
      pAdaptiveParam, Ipp16s* pDstOpenLoopLag, Ipp16s* pToneFlag, Ipp16s*
      pDstOpenLoopGain, Ipp16s* pSrcDstPrevPitchLag, Ipp16s* pSrcDstLagSwitcher,
      int len );
```

Parameters

<i>pSrcWgtSpch</i>	Pointer to a 320-element vector containing perceptually weighted speech.
<i>pSrcFltWgtSpch</i>	Pointer to a 320-element vector containing filtered through high-pass filter perceptually weighted speech.
<i>pPrevMidPitchLag</i>	Pointer to the median filtered pitch lag of the 5 previous voiced speech half-frames.
<i>pAdaptiveParam</i>	Pointer to the adaptive parameter.
<i>pDstOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags.
<i>pToneFlag</i>	Pointer to the tone flag for the VAD module.
<i>pDstOpenLoopGain</i>	Pointer to a 2-element vector containing optimal open-loop pitch gains.
<i>pSrcDstPrevPitchLag</i>	Pointer to the five-element vector that contains the pitch lags associated with the five most recent voiced speech half-frames.
<i>pSrcDstLagSwitcher</i>	Switches lag weighting on and off.
<i>len</i>	Length of the frame.

Description

This function is declared in `ippsc.h` file. The open-loop pitch (OLP) search function `ippsOpenLoopPitchSearch_AMRWB_16s` extracts a pitch estimate from a weighted version of the input speech. The OLP search details are as follows:

1. Compute a windowed autocorrelation of weighted speech, that is,

$$R(k) = w(k) \times \sum_{n=0}^{len} sw(n) \times sw(n-k), \quad 17 \leq k \leq 115$$

where the sequence $sw(n)$ contains weighted speech, and w is the weighting function given by

$$w(k) = wl(k) \times wn(k)$$

In this weighting function, wl emphasizes low pitch and is defined in terms of the table cw , while wn is a sequence of neighboring emphasis lag coefficients associated with the previous frame, that is,

$$wn(k) = \begin{cases} cw(T_{old} - k + 98), & pSrcDstLagSwitcher = 1 \\ 1.0, & otherwise \end{cases}$$

and the parameter T_{old} is the median filtered pitch lag of 5 previous voiced speech half frames.

The estimated open-loop pitch lag is the value k that maximizes $R(k)$. It is denoted by T_{op} .

2. Compute the optimal open-loop gain using the relation

$$g = \frac{\sum_{k=0}^{len-1} sw(k) sw(k - T_{op})}{\sqrt{\sum_{k=0}^{len-1} s^2 w(k) \sum_{k=0}^{len-1} s^2 w(k - T_{op})}}$$

In addition, adaptive parameter is re-calculated

$$v = \begin{cases} 1.0, & g > 0.6 \\ 0.9v, & otherwise \end{cases}$$

and

$$pSrcDstLagSwitcher = \begin{cases} 1, & v > 0.8 \\ 0, & otherwise \end{cases}$$

If $g > 0.6$, the previous pitch lag buffer and median pitch lag of the previous pitch lags are updated.

Update the tone flag as follows:

toneflag = *toneflag* >> 1

and if $g > 0.65$

toneflag = *toneflag* | 0x4000.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcWgtSpch</i> , <i>pSrcFltWgtSpch</i> , <i>pPrevMidPitchLag</i> , <i>pAdaptiveParam</i> , <i>pDstOpenLoopLag</i> , <i>pToneFlag</i> , <i>pDstOpenLoopGain</i> , <i>pSrcDstPrevPitchLag</i> or <i>pSrcDstLagSwitcher</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

Filtering Functions

HighPassFilterGetSize_AMRWB

Returns the size of the high-pass filter state memory.

Syntax

```
IppStatus ippHighPassFilterGetSize_AMRWB_16s (int order, int* pDstSize);
```

Parameters

<i>order</i>	The order of high-pass filter (orders of 2 or 3 are currently supported).
<i>pDstSize</i>	Pointer to the output value of the memory size.

Description

This function is declared in `ippsc.h` file. It returns the size of memory needed for proper operation of the high-pass filter of the given *order*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsRangeErr</code>	Indicates an error when <i>order</i> is not equal to either 2 or 3.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDstSize</i> pointer is <code>NULL</code> .

HighPassFilterInit_AMRWB

Initializes the state memory of high-pass filter.

Syntax

```
IppStatus ippHighPassFilterInit_AMRWB_16s(Ipp16s* pFilterCoeffA, Ipp16s*
    pFilterCoeffB, int order, IppsHighPassFilterState_AMRWB_16s* pState);
```

Parameters

<i>order</i>	The order of high-pass filter; orders of 2 and 3 are currently supported.
<i>pFilterCoeffA</i>	Pointer to the vector of <i>order</i> size containing the IIR part of the filter coefficients.
<i>pFilterCoeffB</i>	Pointer to the vector of <i>order</i> size containing the FIR part of the filter coefficients.
<i>pState</i>	Pointer to the memory supplied for filtering.

Description

This function is declared in `ippsc.h` file. It sets the coefficients and initializes the state memory of the high-pass filter.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFilterCoeffA</i> , <i>pFilterCoeffB</i> or <i>pState</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>order</i> is not equal to either 2 or 3.

HighPassFilter_AMRWB

Performs high-pass filtering.

Syntax

```
IppStatus ippsHighPassFilter_AMRWB_16s_Sfs (const Ipp16s* pSrc, Ipp16s* pDst,
      int len, IppsHighPassFilterState_AMRWB_16s* pState, int scaleFactor);
IppStatus ippsHighPassFilter_AMRWB_16s_ISfs (Ipp16s* pSrcDst, int len,
      IppsHighPassFilterState_AMRWB_16s* pState, int scaleFactor);
IppStatus ippsHighPassFilter_Direct_AMRWB_16s (const Ipp16s* pSrcCoeff, const
      Ipp16s* pSrc, Ipp16s* pDst, int len, int borderMode);
```

Parameters

<i>pSrc</i>	Pointer to the vector containing input signal.
<i>pDst</i>	Pointer to the vector containing filtered signal.
<i>pSrcDst</i>	Pointer to the input/output vector.
<i>pState</i>	Pointer to the memory supplied for filtering.
<i>scaleFactor</i>	Scale factor of the input signal.
<i>pSrcCoeff</i>	Pointer to a 2-element vector containing the coefficients of the filter.
<i>len</i>	Length of the input and output vectors.
<i>borderMode</i>	if <i>borderMode</i> = 0 then <i>pSrc</i> [-1] and <i>pSrc</i> [<i>len</i>] are equal to zero; otherwise these values must be given.

Description

These functions are declared in `ippsc.h` file.

ippsHighPassFilter_AMRWB_16s. This function performs the high-pass filtering of the input signal by the following transfer function:

$$H_h(z) = \frac{b_0 + b_1 z^{-1} \dots + b_{order-1} z^{-order}}{a_0 + a_1 z^{-1} \dots + a_{order-1} z^{-order}}$$

Currently, only $a_0 = 1.0$ is supported. The scale factor is used to update the filter memory according to the input signal.

ippsHighPassFilter_Direct_AMRWB_16s. This function performs the high-pass filtering of the input signal by the following transfer function:

$$H_h(z) = \frac{a_0}{1 + a_1 z^{-1}}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pSrcCoeff</i> , <i>pSrcDst</i> , <i>pState</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

Preemphasize_AMRWB

Computes pre-emphasis of a speech signal.

Syntax

```
IppStatus ippsPreemphasize_AMRWB_16s_ISfs (Ipp16s gamma, Ipp16s* pSrcDst, int
    len, int scaleFactor, Ipp16s* pMem);
```

Parameters

<i>gamma</i>	Filter coefficient.
<i>pSrcDst</i>	Pointer to the input/output vector.
<i>scaleFactor</i>	Scale factor for the result.
<i>len</i>	Length of the input/output vector.

pMem Pointer to the filter memory of length 1.

Description

This function is declared in `ippsc.h` file. It computes pre-emphasis of the input speech according to the difference signal pre-emphasis equation:

$$H(z) = 1 - (\gamma \cdot z^{-1})$$

The memory value *pMem*[0] is updated by *pSrcDst*[n-1]. For proper use of this function in AMR WB codec, the memory value must be initialized to zero in the user program.

Return Values

ippStsNoErr Indicates no error.

ippStsNullPtrErr Indicates an error when the *pSrcDst* or *pMem* pointer is NULL.

ippStsSizeErr Indicates an error when *len* is less than or equal to zero.

ippStsScaleRangeErr Indicates an error when *scaleFactor* < 0 or *scaleFactor* > 15.

Deemphasize_AMRWB

Performs de-emphasis filtering.

Syntax

```
IppStatus ippDeemphasize_AMRWB_NR_16s_I(Ipp16s gamma, Ipp16s* pSrcDst, Ipp16s
    len, Ipp16s* pMem);

IppStatus ippDeemphasize_AMRWB_32s16s(Ipp16s gamma, const Ipp32s* pSrc,
    Ipp16s* pDst, int len, Ipp16s* pMem);
```

Parameters

gamma De-emphasis factor.

pSrc Pointer to the input vector.

pDst Pointer to the output vector.

pSrcDst Pointer to the input/output vector.

<i>len</i>	Length of the input/output vector.
<i>pMem</i>	Pointer to the filter memory element.

Description

This function is declared in `ippsc.h` file. It performs de-emphasis of the input synthesized signal by filtering it through the filter with the following transfer function:

$$H(z) = \frac{1}{1 - \gamma z^{-1}}$$

The initial memory of the filter must be set to zero. The memory value `pMem[0]` is updated by `pDst[len-1]` or `pSrcDst[len-1]`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , <i>pSrcDst</i> or <i>pMem</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

SynthesisFilter_AMRWB

Reconstructs the speech signal from LP coefficients and residuals.

Syntax

```
IppStatus ippSsynthesisFilter_AMRWB_16s32s_I(const Ipp16s* pSrcLpc, int order,
const Ipp16s* pSrcExc, Ipp32s* pSrcDstSignal, int len);
```

Parameters

<i>pSrcLpc</i>	Pointer to the LP coefficients vector $a_0, a_1, \dots, a_{order}$
<i>order</i>	Order of LP filter.
<i>pSrcExc</i>	Pointer to the excitation vector.
<i>pSrcDstSignal</i>	Pointer to the synthesized and updated speech.

len Length of the filter.

Description

This function is declared in `ippsc.h` file. It computes the filter given by:

$$H(z) = \frac{1}{a_0 + \sum_{i=1}^{order} a_i z^{-i}}$$

Return Values

<code>IppStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pSrcLpc</i> , <i>pSrcExc</i> or <i>pSrcDstSignal</i> pointer is NULL.
<code>IppStsSizeErr</code>	Indicates an error when <i>len</i> or <i>order</i> is less than or equal to zero.

Discontinuous Transmission (DTX) Functions

VADGetSize_AMRWB

Returns the size of the VAD module state memory.

Syntax

```
IppStatus ippVADGetSize_AMRWB_16s (int* pDstSize);
```

Parameters

pDstSize Pointer to the output value of the memory size.

Description

This function is declared in `ippsc.h` file. It returns the size of memory needed for proper operation of the VAD module in the AMR WB codec .

Return Values

<code>IppStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`IppStsNullPtrErr` Indicates an error when the `pDstSize` pointer is NULL.

VADInit_AMRWB

Initializes the VAD module state memory.

Syntax

```
IppStatus ippsVADInit_AMRWB_16s(IppsVADState_AMRWB_16s* pState);
```

Parameters

`pState` Pointer to the VAD memory.

Description

This function is declared in `ippsc.h` file. It initializes the state of VAD (Voice Activity Decision) module of the AMR WB codec. The structure `IppsVADState_AMRWB_16s` contains the VAD history that must be initialized prior to the use of the function `ippsVAD_AMRWB_16s`.

Return Values

`ippStsNoErr` Indicates no error.

`IppStsNullPtrErr` Indicates an error when the `pState` pointer is NULL.

VAD_AMRWB

Performs VAD in AMR WB encoder.

Syntax

```
IppStatus ippsVAD_AMRWB_16s(const Ipp16s* pSrcSpch, IppsVADState_AMRWB_16s*  
pSrcDstVadState, Ipp16s* pToneFlag, Ipp16s* pVadFlag);
```

Parameters

`pSrcSpch` Pointer to the input speech signal, of length 320.

`pSrcDstVadState` Pointer to the VAD memory.

<i>pToneFlag</i>	Pointer to the output tone flag.
<i>pVadFlag</i>	Pointer to the VAD flag of this frame.

Description

This function is declared in `ippsc.h` file. The function `ippsVAD_AMRWB_16s` implements the Voice Activity Decision (VAD) functionality of the AMR WB encoder. This function is used to indicate whether the input speech frame contains active speech or some other audio signal such as silence or music.

The structure `IppsVADState_AMRWB_16s` contains the VAD history and is updated in the function. This structure should be initialized in advance by using the function `ippsVADInit_AMRWB_16s`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pSrcSpch</i> , <i>pSrcDstVadState</i> , <i>pToneFlag</i> or <i>pVadFlag</i> pointer is NULL.

Codebook Search Functions

AlgebraicCodebookSearch_AMRWB

Performs the fixed (algebraic) codebook search.

Syntax

```
IppStatus ippsAlgebraicCodebookSearch_AMRWB_16s(const Ipp16s* pSrcFixedTarget,
const Ipp16s* pSrcLtpResidual, Ipp16s* pSrcDstImpulseResponse, Ipp16s*
pDstFixedVector, Ipp16s* pDstFltFixedVector, IppSpchBitRate mode, Ipp16s*
pDstIndex);
```

Parameters

<i>pSrcFixedTarget</i>	Target vector.
<i>pSrcLtpResidual</i>	Long term prediction residual.

<i>pSrcDstImpulseResponse</i>	Impulse response of the weighted synthesis filter; <i>pSrcDstImpulseResponse</i> [-L_subfr..-1] must be set to zero.
<i>pDstFixedVector</i>	Innovative codebook.
<i>pDstFltFixedVector</i>	Filtered fixed codebook excitation.
<i>mode</i>	Coder mode.
<i>pDstIndex</i>	Indexes of the pulses.

Description

This function is declared in `ippsc.h` file. It performs the following steps:

1. Computes backward filtered target vector **d** by

$$d(n) = \sum_{i=n}^{63} x_2(i) \times h(i-n), \quad 0 \leq n \leq 63$$

where $x_2(n)$ is the fixed target signal used for fixed codebook search.

2. Computes the signal $b(n)$, which is used for presetting the pulse amplitudes, as:

$$b(n) = \sqrt{\frac{\sum_{i=0}^{63} d(i)d(i)}{\sum_{i=0}^{63} res_{LTP}(i)res_{LTP}(i)}} res_{LTP}(n) + \alpha d(n), \quad 0 \leq n \leq 63$$

The scaling factor α controls the amount of dependence of the reference signal on $d(n)$, it is lowered as the bit rate is increased. Here $\alpha=2$ for 6.6 and 8.85 modes; $\alpha=1$ for 12.65, 14.25 and 15.85 modes; $\alpha=0.8$ for 18.25 mode; $\alpha=75$ for 19.25 mode; and $\alpha=0.5$ for 23.05 and 23.85 modes.

3. Calculates the symmetric Toeplitz matrix **Φ** for each mode. The element of this matrix is computed by

$$\Phi(i,j) = \sum_{n=j}^{63} h(n-i)h(n-j), \quad i = 0, \dots, 63, j = 0, \dots, 63$$

4. Searches signed pulses in appropriate positions and encodes them. The pulse positions for different modes are in the tables below:

Modes 23.05 and 23.85:

Track	Pulse	Positions
1	$i_0, i_4, i_8, i_{12}, i_{16}, i_{20}$	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5, i_9, i_{13}, i_{17}, i_{21}$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	$i_2, i_6, i_{10}, i_{14}, i_{18}, i_{22}$	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	$i_3, i_7, i_{11}, i_{15}, i_{19}, i_{23}$	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 19.85:

Track	Pulse	Positions
1	$i_0, i_4, i_8, i_{12}, i_{16}$	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5, i_9, i_{13}, i_{17}$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6, i_{10}, i_{14}	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7, i_{11}, i_{15}	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 18.25:

Track	Pulse	Positions
1	i_0, i_4, i_8, i_{12}	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60

2	i_1, i_5, i_9, i_{13}	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6, i_{10}, i_{14}	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7, i_{11}, i_{15}	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 15.85:

Track	Pulse	Positions
1	i_0, i_4, i_8	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	i_1, i_5, i_9	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6, i_{10}	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7, i_{11}	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 14.25:

Track	Pulse	Positions
1	i_0, i_4, i_8	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	i_1, i_5, i_9	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 12.65:

Track	Pulse	Positions
1	i_0, i_4	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	i_1, i_5	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 8.85:

Track	Pulse	Positions
1	i_0	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	i_1	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 6.60:

Track	Pulse	Positions
1	i_0	0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62

2	i_1	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63
---	-------	--

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcFixedTarget</i> , <i>pSrcLtpResidual</i> , <i>pSrcDstImpulseResponse</i> , <i>pDstFixedVector</i> , <i>pDstFltFixedVector</i> or <i>pDstIndex</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

AlgebraicCodebookDecode_AMRWB

Decodes the fixed (algebraic) codebook indexes.

Syntax

```
IppStatus ippAlgebraicCodebookDecode_AMRWB_16s (const Ipp16s* pSrcIdxs,  
Ipp16s* pDstFixedCode, IppSpchBitRate mode);
```

Parameters

<i>pSrcIdxs</i>	Algebraic codebook indexes.
<i>pDstFixedCode</i>	Pointer to the algebraic code vector of length 64.
<i>mode</i>	Coder mode.

Description

This function is declared in `ippsc.h` file. The function decodes the fixed codebook vector from the received fixed codebook index.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcIdxs</i> or <i>pDstFixedCode</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not an element of the enumerated type <code>IppSpchBitRate</code> appropriate to AMR WB.

AdaptiveCodebookGainCoeff_AMRWB

Computes the adaptive codebook gain.

Syntax

```
IppStatus ippAdaptiveCodebookGainCoeff_AMRWB_16s(const Ipp16s*
    pSrcAdptTarget, const Ipp16s* pSrcFltAdptVector, Ipp16s* pAdptGainCoeffs,
    Ipp16s* pResultAdptGain, int len);
```

Parameters

<i>pSrcAdptTarget</i>	Pointer to the adaptive-codebook vector.
<i>pSrcFltAdptVector</i>	Pointer to the input filtered adaptive-codebook vector.
<i>pResultAdptGain</i>	Pointer to the adaptive-codebook gain in the length of 1.
<i>pAdptGainCoeffs</i>	Pointer to the output vector in the length 4; represents the adaptive-codebook gain as a fraction.
<i>len</i>	Length of vectors.

Description

This function is declared in `ippsc.h` file. The adaptive codebook gain is found by:

$$g_p = \frac{\sum_{n=0}^{len-1} x(n)y(n)}{\sqrt{\sum_{n=0}^{len-1} y(n)y(n)}}$$

, bounded by

$$0 \leq g_p \leq 1.2$$

Additionally, the function returns the gain in a different representation given by the following formula:

$$g_p = \frac{c_{xy} 2^{exp_{xy}}}{c_{yy} 2^{exp_{yy}}}, 1/2 \leq c_{xy} / c_{yy} \leq 1$$

The mantissas C_{xy} , C_{yy} and exponents exp_{xy} , exp_{yy} for both the normalized denominator and normalized numerator are returned in the `pAdptGainCoeffs` vector:

```
pAdptGainCoeffs[0] = Cyy
pAdptGainCoeffs[1] = expyy
pAdptGainCoeffs[2] = Cxy
pAdptGainCoeffs[3] = expxy
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcAdptTarget</code> , <code>pSrcFltAdptVector</code> , <code>pResultAdptGain</code> or <code>pAdptGainCoeffs</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

AdaptiveCodebookSearch_AMRWB

Performs the adaptive codebook search.

Syntax

```
IppStatus ippsAdaptiveCodebookSearch_AMRWB_16s(const Ipp16s* pSrcAdptTarget,
const Ipp16s* pSrcImpulseResponse, const Ipp16s* pSrcOpenLoopLag, Ipp16s*
pPitchLag, Ipp16s* pPitchLagBounds, Ipp16s* pSrcDstExcitation, Ipp16s*
pFracPitchLag, Ipp16s* pAdptIndex, int subFrame, IppSpchBitRate mode);
```

Parameters

<i>pSrcAdptTarget</i>	Pointer to the 64-element adaptive target signal vector.
<i>pSrcImpulseResponse</i>	Pointer to the 64-element impulse response of the weighted synthesis filter.
<i>pSrcOpenLoopLag</i>	Pointer to a two-element vector of the open-loop pitch lags.
<i>pPitchLag</i>	Pointer to the output previous integral pitch lag.
<i>pPitchLagBounds</i>	Pointer to the bounds of the output previous integral pitch lag.
<i>pSrcDstExcitation</i>	Pointer to the input/output 321-element excitation vector.
<i>pFracPitchLag</i>	Pointer to the output fractional pitch lag obtained during the adaptive codebook search.
<i>pAdptIndex</i>	Pointer to the coded closed-loop pitch index.
<i>subFrame</i>	Subframe number.
<i>mode</i>	Coder mode.

Description

This function is declared in `ipps.h` file. The function performs the adaptive codebook search. The adaptive codebook search consists of a closed-loop pitch search followed by computation of an adaptive excitation vector. The adaptive excitation vector is computed by interpolating the past excitation at the fractional pitch lag obtained during the closed-loop pitch search. The adaptive codebook is searched on every subframe. A detailed description of the adaptive codebook search procedure is given next:

1. In the 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s modes, in the first and third

subframes, a fractional pitch delay is used with resolutions **1/4** in the range $\left[34, 127 \frac{3}{4}\right]$, resolutions **1/2** in the range $\left[128, 159 \frac{1}{2}\right]$, and integers only in the range **[160, 231]**. For the second and fourth subframes, a pitch resolution of **1/4** is always used in the range $\left[T_1 - 8, T_1 + 7 \frac{3}{4}\right]$, where T_1 is the nearest integer to the fractional pitch lag of the previous (1st or 3rd) subframe.

In 6.60 kbit/s mode, in the first subframe, a fractional pitch delay is used with resolutions **1/2** in the range $\left[34, 91 \frac{1}{2}\right]$, and integers only in the range **[92, 231]**. For the second, third and fourth subframes, a pitch resolution of **1/2** is always used in the range $\left[T_1 - 8, T_1 + 7 \frac{3}{4}\right]$, where T_1 is the nearest integer to the fractional pitch lag of the first subframe.

Closed-loop pitch analysis is performed around the open-loop pitch estimates on a subframe basis. In 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s modes, in the first (and third) subframe the range $T_{op} \pm 7$, bounded by 34...231, is searched. In 6.60 kbit/s mode, in the first subframe the range $T_{op} \pm 7$, bounded by 34...231, is searched. For all the modes, for the other subframes, closed-loop pitch analysis is performed around the integer pitch selected in the previous subframe, as described above. In 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s modes, the pitch delay is encoded with 9 bits in the first and third subframes and the relative delay of the other subframes is encoded with 6 bits. In 8.85 kbit/s mode, the pitch delay is encoded with 8 bits in the first and third subframes and the relative delay of the other subframes is encoded with 5 bits. In 6.60 kbit/s mode, the pitch delay is encoded with 8 bits in the first subframe and the relative delay of the other subframes is encoded with 5 bits.

2. The closed loop pitch search is performed by minimizing the mean-square weighted error between the original and synthesized speech. This is achieved by maximizing the term

$$R(k) = \frac{\sum_{n=0}^{63} x(n) y_k(n)}{\sqrt{\sum_{n=0}^{63} y_k(n) y_k(n)}}$$

where $x(n)$ is the target signal and $y_k(n)$ the past filtered excitation at delay k (past excitation convolved with $h(n)$). Note that the search range is limited around the open-loop pitch as explained earlier.

The convolution $y_k(n)$ is computed for the first delay in the searched range, and for the other delays it is updated using the recursive relation

$$y_k(n) = y_{k-1}(n-1) + u(-k)h(n)$$

where

$$u(n), n = -(231+17), \dots, 63$$

is the excitation buffer. Note that in search stage, the samples $u(n)$, $n = 0, \dots, 63$ are not known, and they are needed for pitch delays less than 64.

3. Let

$$B_{LP} = 0.18z + 0.64 + 0.18z^{-1}$$

be the low-pass filter. Thus the adaptive codebook vector $v(n)$ is computed as

$$v(n) = \sum_{i=1}^l b_{LP}(i+1)v'(n+i)$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcAdptTarget</i> , <i>pSrcImpulseResponse</i> , <i>pSrcOpenLoopLag</i> , <i>pPitchLag</i> , <i>pPitchLagBounds</i> , <i>pSrcDstExcitation</i> , <i>pFracPitchLag</i> or <i>pAdptIndex</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> < 0 or <i>subFrame</i> > 3.

AdaptiveCodebookDecodeGetSize_AMRWB

Queries the memory length of the adaptive codebook decode module.

Syntax.

```
IppStatus ippsAdaptiveCodebookDecodeGetSize_AMRWB_16s(int* pDstSize);
```

Parameters

<i>pDstSize</i>	Pointer to the output parameter to store the adaptive codebook decode module length in bytes.
-----------------	---

Description

This function is declared in `ippsc.h` file. The function reports the size of memory needed for proper operation of the adaptive codebook decode module.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDstSize</i> pointer is NULL.

AdaptiveCodebookDecodeInit_AMRWB

Initializes the adaptive codebook decode module memory.

Syntax

```
IppStatus ippsAdaptiveCodebookDecodeInit_AMRWB_16s(  
    IppsAdaptiveCodebookDecodeState_AMRWB_16s* pState);
```

Parameters

<i>pState</i>	Pointer to the adaptive codebook decode module memory.
---------------	--

Description

This function is declared in `ippsc.h` file. The function initializes the adaptive codebook decode module memory.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pState</code> pointer is NULL.

AdaptiveCodebookDecodeUpdate_AMRWB

Updates the adaptive codebook decode module memory.

Syntax

```
IppStatus ippAdaptiveCodebookDecodeUpdate_AMRWB_16s(Ipp16s valIntPitchLag,
    Ipp16s valPitchGain, IppsAdaptiveCodebookDecodeState_AMRWB_16s* pState);
```

Parameters

<code>valIntPitchLag</code>	Pointer to the integral pitch lag.
<code>valPitchGain</code>	Pitch gain.
<code>pState</code>	Pointer to the adaptive codebook decode module memory.

Description

This function is declared in `ippsc.h` file. The function updates the adaptive codebook decode module memory with the given pitch lag and pitch gain.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pState</code> pointer is NULL.

AdaptiveCodebookDecode_AMRWB

Performs the adaptive codebook decoding.

Syntax

```
IppStatus ippsAdaptiveCodebookDecode_AMRWB_16s(int valAdptIndex, Ipp16s*
    pResultFracPitchLag, Ipp16s* pSrcDstExcitation, Ipp16s* pPitchLag, Ipp16s*
    pPitchLagBounds, int subFrame, int bfi, int unusableFrame, IppSpchBitRate
    mode, IppsAdaptiveCodebookDecodeState_AMRWB_16s* pState);
```

Parameters

<i>valAdptIndex</i>	Adaptive codebook index.
<i>pResultFracPitchLag</i>	Pointer to the fractional pitch lag obtained during the adaptive codebook search.
<i>pSrcDstExcitation</i>	Pointer to the 321-element excitation vector.
<i>pPitchLag</i>	Pointer to the integral pitch lag.
<i>pPitchLagBounds</i>	Pointer to the previous integral pitch lag bounds.
<i>subFrame</i>	Subframe number.
<i>bfi</i>	Bad frame indicator. Value "0" signifies a good frame; any other value signifies a bad frame.
<i>unusableFrame</i>	Value "0" signifies a lost frame with content that can not be used at all.
<i>mode</i>	Encoding mode.
<i>pState</i>	Pointer to the memory of the adaptive codebook decode module.

Description

This function is declared in `ippsc.h` file. It performs the following steps:

- 1.If no errors are detected on the current frame, the function extracts integer and fractional pitch lags from the adaptive codebook indices.
- 2.If errors are detected, the function recovers the integer pitch either from the previous integer pitch or the LTP-lag. See clause 6.2.3.4 in 3GPP TS 26.191 V5.1.0.

3. To obtain the adaptive codebook vector, the function `ippsAdaptiveCodebookDecode_AMRWB_16s` applies the same adaptive codebook computation procedure as described in the function [ippsAdaptiveCodebookSearch_AMRWB_16s](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> , <i>pResultFracPitchLag</i> , <i>pSrcDstExcitation</i> , <i>pPitchLag</i> or <i>pPitchLagBounds</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> < 0 or <i>subFrame</i> > 3.

Quantization Functions

ISFQuant_AMRWB

Quantizes the ISF.

Syntax

```
IppStatus ippsISFQuant_AMRWB_16s(const Ipp16s* pSrcIsf, Ipp16s*
    pSrcDstResidual, Ipp16s* pDstQIsf, Ipp16s* pDstQIsfIndex, IppSpchBitRate
    mode);
```

Parameters

<i>pSrcIsf</i>	Pointer to the unquantized 16-element ISF vector.
<i>pSrcDstResidual</i>	Pointer to the 16-element quantized ISF residual from the previous frame.
<i>pDstQIsf</i>	Pointer to the quantized 16-element ISF vector.
<i>pDstQIsfIndex</i>	Pointer to the 7-element vector of quantized ISF indices. For 6.60 Kbps frames, only the first five elements contain valid indices; for all other bit rates, all seven elements contain valid data.
<i>mode</i>	Coder mode.

Description

This function is declared in `ippsc.h` file. It applies a first order moving-average (MA) prediction and quantifies the residual ISF vector using a combination of split vector quantization (SVQ) and multistage vector quantization (MSVQ).

The prediction and quantization are performed as follows. Let $z(n)$ denote the mean-removed ISF vector at frame n .

The prediction residual vector $r(n)$ is given by:

$$r(n) = z(n) - p(n)$$

where $p(n)$ is the predicted LSF vector at frame n . First order moving-average (MA) prediction is used where:

$$p(n) = \frac{1}{3} \hat{r}(n-1)$$

where $\hat{r}(n-1)$ is the quantized residual vector at the past frame.

The ISF residual vector r is quantized using split-multistage vector quantization S-MSVQ. The vector is split into two subvectors $r_1(n)$ and $r_2(n)$ of dimensions 9 and 7, respectively. The two subvectors are quantized in two stages. In the first stage $r_1(n)$ is quantized with 8 bits and $r_2(n)$ with 8 bits.

For 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s modes, the quantization error vectors are split in the next stage into 3 and 2 subvectors, respectively.

For 6.60 kbit/s mode, the quantization error vectors

$$r_i^{(2)} = r - \hat{r}_i, i = 1, 2$$

are split in the next stage into 2 and 1 subvectors, respectively. A squared error ISF distortion measure is used in the quantization process. In general, for an input ISF or error residual subvector $r_i, i = 1, 2$ a quantized vector at index

$$k, \hat{r}_i^k$$

the quantization is performed by finding the index k which minimizes

$$E = \sum_{i=m}^n \left[r_i - \hat{r}_i^k \right]^2$$

where m and n are the first and last elements of the subvector.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcIsf</code> , <code>pSrcDstResidual</code> , <code>pDstQIsf</code> or <code>pDstQIsfIndex</code> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <code>mode</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

ISFQuantDecode_AMRWB

Decodes quantized ISFs from the received codebook index.

Syntax

```
IppStatus ippISFQuantDecode_AMRWB_16s(const Ipp16s* pSrcIdxs, Ipp16s*
    pDstQntIsf, Ipp16s* pSrcDstResidual, const Ipp16s* pSrcPrevQntIsf, Ipp16s*
    pSrcDstIsfMemory, int bfi, IppSpchBitRate mode);
```

Parameters

<code>pSrcIdxs</code>	Pointer to the seven-element vector containing codebook indices of the quantized LSPs.
<code>pSrcDstResidual</code>	Pointer to the 16-element quantized ISF residual from the previous frame.
<code>pSrcPrevQntIsf</code>	Pointer to the 16-element quantized ISF vector from the previous frame.
<code>pSrcDstIsfMemory</code>	Pointer to the 64-element vector containing four subframe ISF sets.
<code>pDstQntIsf</code>	Pointer to a 16-element destination vector containing quantized ISF in frequency domain (0..0.5).

<i>bfi</i>	Bad frame indicator: value "0" signifies a good frame, all other values signify a bad frame.
<i>mode</i>	Coder mode.

Description

This function is declared in `ippsc.h` file. The function decodes quantized ISFs from the received codebook index if the errors are not detected on the received frame. Otherwise, the function recovers the quantized ISFs from previous quantized ISFs using linear interpolation.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcIdxs</i> , <i>pSrcDstResidual</i> , <i>pSrcPrevQntIsf</i> , <i>pSrcDstIsfMemory</i> or <i>pDstQntIsf</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

ISFQuantDTX_AMRWB

Quantizes the ISF coefficient vector in case of DTX mode.

Syntax

```
IppStatus ippISFQuantDTX_AMRWB_16s(const Ipp16s* pSrcIsf, Ipp16s* pDstQntIsf,
Ipp16s* pDstIdxs);
```

Parameters

<i>pSrcIsf</i>	Pointer to the unquantized 16-element ISF vector in the frequency domain (0..0.5).
<i>pDstQntIsf</i>	Pointer to the 16-element quantized ISF vector.
<i>pDstIdxs</i>	Pointer to the 5-element vector quantization indices.

Description

This function is declared in `ippsc.h` file. The function quantizes averaged ISF coefficient vector using the comfort noise ISF quantization tables.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcIsf</i> , <i>pDstQntIsf</i> or <i>pDstIdxs</i> pointer is NULL.

ISFQuantDecodeDTX_AMRWB

Decodes quantized ISFs in case of DTX mode.

Syntax

```
IppStatus ippISFQuantDecodeDTX_AMRWB_16s(const Ipp16s* pSrcIdxs, Ipp16s*
pDstQntIsf);
```

Parameters

<i>pSrcIdxs</i>	Pointer to the 5-element vector quantization indices.
<i>pDstQntIsf</i>	Pointer to the 16-element ISF vector in the frequency domain (0..0.5) .

Description

This function is declared in `ippsc.h` file. The function decodes quantized ISFs from the received codebook index in case of DTX mode.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcIdxs</i> or <i>pDstQntIsf</i> pointer is NULL.

GainQuant_AMRWB

Quantizes the adaptive codebook gains.

Syntax

```
IppStatus ippsGainQuant_AMRWB_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s*
    pSrcFltAdptVector, int valFormat, const Ipp16s* pSrcFixedVector, const
    Ipp16s* pSrcFltFixedVector, const Ipp16s* pSrcCorr, Ipp16s*
    pSrcDstEnergyErr, Ipp16s* pSrcDstPitchGain, int* pDstCodeGain, int
    valClipFlag, Ipp16s* pDstQGainIndex, int lenSrc, IppSpchBitRate mode);
```

Parameters

<i>pSrcAdptTarget</i>	Pointer to the input target vector $x(n)$.
<i>pSrcFltAdptVector</i>	Pointer to the input filtered adaptive codebook vector $y(n)$.
<i>valFormat</i>	Format of the <i>pSrcAdptTarget</i> and <i>pSrcFltAdptVector</i> vectors.
<i>pSrcFixedVector</i>	Pointer to the input pre-filtered codebook contribution $c(n)$.
<i>pSrcFltFixedVector</i>	Pointer to the input filtered codebook vector $z(n)$.
<i>pSrcCorr</i>	Pointer to the vector of correlations between the <i>pSrcAdptTarget</i> , <i>pSrcFltAdptVector</i> , <i>pSrcFltFixedVector</i> vectors.
<i>pSrcDstEnergyErr</i>	Pointer to the input/output energy error vector for 4 previous subframes.
<i>pSrcDstPitchGain</i>	Pointer to the input/output pitch gain.
<i>pDstCodeGain</i>	Pointer to the output code gain.
<i>valClipFlag</i>	If <i>valClipFlag</i> = 1, then limit gain pitch to 1.0.
<i>pDstQGainIndex</i>	Pointer to the output codebook indexes found.
<i>lenSrc</i>	Length of the input vectors.
<i>mode</i>	Encoding mode.

Description

This function is declared in `ippsc.h` file. The adaptive codebook gain (pitch gain) and the fixed (algebraic) codebook gain are vector quantized using a 6-bit codebook for modes 8.85 and 6.60 kbit/s and using a 7-bit codebook for all the other modes.

The functionality is as follows:

1. Calculate the average energy minus the mean energy of the fixed-codebook contribution:

$$E = 10 \log \left(\frac{1}{lenSrc} \sum_{i=0}^{lenSrc-1} c^2(i) \right)$$

Then calculate

$$E_b - E = 30(dB) - E$$

2. Calculate the predicted energy from prediction error of the previous subframes:

$$\tilde{E}^m = \sum_{i=1}^4 b_i \hat{U}^{(m-1)}$$

where \hat{U}^m is calculated by the equation $\hat{U}^m = E - \tilde{E}^m$ and $[b1 \ b2 \ b3 \ b4] = [0.5, 0.4, 0.3, 0.2]$ are the MA prediction coefficients.

3. Calculate g_c (the fixed-codebook gain) by the equation

$$g_c = 10^{0.05(E^m + E_b - E)}$$

4. The gain codebook search is performed by minimizing the mean-square of the weighted error between original and reconstructed speech which is given

$$E = x^t x + g_p^2 y^t y + g_c^2 z^t z - 2g_p x^t y - 2g_c x^t z + 2g_p g_c y^t z$$

where x is the target vector, y is the filtered adaptive codebook vector, and z is the filtered fixed codebook vector. (Each gain vector in the codebook also has an element representing the quantized energy prediction error.) The quantized energy prediction error associated with the

chosen gains is used to update \tilde{U}^m . In the search, only the 64 codevectors that are closest to the unquantized pitch gain g_p are taken into account.

Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcAdptTarget</i> , <i>pSrcFltAdptVector</i> , <i>pSrcFixedVector</i> , <i>pSrcFltFixedVector</i> , <i>pSrcCorr</i> , <i>pSrcDstEnergyErr</i> , <i>pSrcDstPitchGain</i> , <i>pDstCodeGain</i> or <i>pDstQGainIndex</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>lenSrc</i> is less than or equal to zero.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

DecodeGain_AMRWB

Decodes adaptive and fixed-codebook gains.

Syntax

```
IppStatus ippDecodeGain_AMRWB_16s(int valQIndex, Ipp32s valEnergy, Ipp16s*  
    pDstPitchGain, int* pDstCodeGain, int bfi, int prevBfi, Ipp16s*  
    pSrcDstPastEnergy, Ipp16s* pPrevCodeGain, Ipp16s* pSrcDstPastCodeGain,  
    IppSpchBitRate mode);
```

Parameters

<i>valQIndex</i>	Index of quantization.
<i>valEnergy</i>	Energy of the code vector.
<i>pDstPitchGain</i>	Pointer to the decoded pitch gain.
<i>pDstCodeGain</i>	Pointer to the decoded code gain.
<i>bfi</i>	Bad frame indicator.
<i>prevBfi</i>	Bad frame indicator of the previous frame.
<i>pSrcDstPastEnergy</i>	Past quantized energies.
<i>pPrevCodeGain</i>	Past code gain.
<i>pSrcDstPastCodeGain</i>	Past code gain for frame erasures.
<i>mode</i>	Decoding mode.

Description

This function is declared in `ippsc.h` file. It performs the following steps.

If there are no errors detected in the frame, the function computes gains as described below:

- 1.The received index gives the fixed codebook gain correction factor $\hat{\gamma}$.
- 2.The predicted energy for every subframe m is computed as

$$\tilde{E}^m = \sum_{i=1}^4 b_i \hat{U}^{(m-1)}$$

- 3.The mean innovation energy is found by

$$E = 10 \log \left(\frac{1}{64} \sum_{i=0}^{63} c^2(i) \right)$$

- 4.The predicted gain g_c' is computed as

$$g_c = 10^{0.05(\tilde{E}^m + E_b - E)}$$

- 5.The quantized fixed codebook gain is given by

$$\hat{g}_c = \hat{\gamma} g_c'$$

In case of frame erasure, the gains are the attenuated versions of the previous gains.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDstPitchGain</code> , <code>pDstCodeGain</code> , <code>pSrcDstPastEnergy</code> , <code>pPrevCodeGain</code> or <code>pSrcDstPastCodeGain</code> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <code>mode</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

EncDTXBuffer_AMRWB

Buffers the ISP coefficients and previous logarithm energy coefficients.

Syntax

```
IppStatus ippsEncDTXBuffer_AMRWB_16s(const Ipp16s* pSrcSpch, const Ipp16s*
    pSrcIsp, Ipp16s* pUpdateIndex, Ipp16s* pSrcDstIspBuffer, Ipp16s*
    pSrcDstLogEnergyBuffer, IppSpchBitRate mode);
```

Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal of length 256.
<i>pSrcIsp</i>	Pointer to the ISP for this frame, of length 16.
<i>pUpdateIndex</i>	Holds the previous memory update index. On output, it contains the current memory update index. Circularly increased between 0 and 7.
<i>pSrcDstIspBuffer</i>	Pointer to the ISP coefficients of eight previous frames, of length 128. On output, points to the ISP coefficients of eight most recent frames (including the current frame).
<i>pSrcDstLogEnergyBuffer</i>	Pointer to the logarithm energy coefficients of eight previous frames, of length 8. On output, points to the logarithm energy coefficients of eight most recent frames (including the current frame).
<i>mode</i>	Bit rate specifier. Values between IPP_SPCHBR_6600 and IPP_SPCHBR_23850 are valid.

Description

The function `ippsEncDTXBuffer_AMRWB` is declared in `ippsc.h` file. This function buffers the ISP coefficients and previous logarithm energy coefficients. The buffered ISP coefficients and energy coefficients will be used for SID frame to extract necessary parameters. The memory update index indicates which part of the buffer will be updated and thus saves the cost of some memory copy operation. The logarithm energy is computed as follows:

$$en_{\log}(i) = \frac{1}{2} \log_2 \left(\frac{1}{256} \sum_{n=0}^{255} pSrcSpch^2(n) \right)$$

After computation, the energy is adjusted according to the current encoder mode.

The function `ippsEncDTXBuffer_AMRWB` behaves similarly to the [ippsEncDTXBuffer_GSMAMR](#) function of the GSM-AMR codec.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcSpch</i> , <i>pSrcIsp</i> , <i>pUpdateIndex</i> , <i>pSrcDstIspBuffer</i> or <i>pSrcDstLogEnergyBuffer</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

DecDTXBuffer_AMRWB

Buffers the ISF coefficients and previous logarithm energy coefficients.

Syntax

```
IppStatus ippsDecDTXBuffer_AMRWB_16s(const Ipp16s* pSrcSpch, const Ipp16s*
    pSrcIsf, Ipp16s* pUpdateIndex, Ipp16s* pSrcDstIsfBuffer, Ipp16s*
    pSrcDstLogEnergyBuffer);
```

Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal of length 256.
<i>pSrcIsf</i>	Pointer to the ISF for this frame, of length 16.
<i>pUpdateIndex</i>	Holds the previous memory update index. On output, it contains the current memory update index. Circularly increased between 0 and 7.

<i>pSrcDstIsfBuffer</i>	Pointer to the ISF coefficients of eight previous frames, of length 128. On output, points to the ISF coefficients of eight most recent frames (including the current frame).
<i>pSrcDstLogEnergyBuffer</i>	Pointer to the logarithm energy coefficients of eight previous frames, of length 8. On output, points to the log energy coefficients of eight most recent frames (including the current frame).

Description

The function `ippsDecDTXBuffer_AMRWB` is declared in `ippsc.h` file.

This function buffers the ISF coefficients and previous logarithm energy coefficients. The buffered ISF coefficients and energy coefficients will be used for SID frame to extract necessary parameters.

The memory update index indicates which part of the buffer will be updated and thus saves the cost of some memory copy operation. The logarithm energy is computed as follows:

$$en_{\log}(i) = \frac{1}{2} \log_2 \left(\frac{1}{256} \sum_{n=0}^{255} pSrcSpch^2(n) \right)$$

The function `ippsDecDTXBuffer_AMRWB` behaves similarly to the [ippsDecDTXBuffer_GSMAMR](#) function of the GSM-AMR codec.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcSpch</i> , <i>pSrcIsf</i> , <i>pUpdateIndex</i> , <i>pSrcDstIsfBuffer</i> or <i>pSrcDstLogEnergyBuffer</i> pointer is NULL.

GSM Full Rate Related Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following the GSM 06.10, 06.11, 06.12, 06.31, and 06.32 recommendations. The list of these functions is given in the following table:

Table 9-6 Intel IPP GSM Full Rate Related Functions

Function Base Name	Operation
<u>RPEQuantDecode_GSMFR</u>	Performs APCM inverse quantization.
<u>Deemphasize_GSMFR</u>	Performs de-emphasis filtering.
<u>ShortTermAnalysisFilter_GSMFR</u>	Performs short-term analysis filtering.
<u>ShortTermSynthesisFilter_GSMFR</u>	Performs short-term synthesis filtering.
<u>HighPassFilter_GSMFR</u>	Filters input speech signal through a high-pass filter.
<u>Schur_GSMFR</u>	Estimates the reflection coefficients by Schur recursion.
<u>WeightingFilter_GSMFR</u>	Calculates the weighting filter.
<u>Preemphasize_GSMFR</u>	Computes pre-emphasis of a speech signal.

RPEQuantDecode_GSMFR

Performs APCM inverse quantization.

Syntax

```
IppStatus ippsRPEQuantDecode_GSMFR_16s (const Ipp16s* pSrc, Ipp16s ampl, Ipp16s  
    amplSfs, Ipp16s* pDst);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [13] of the RPE samples.
<i>ampl</i>	The block amplitude.
<i>amplSfs</i>	Scale factor of the block amplitude.
<i>pDst</i>	Pointer to the output reconstructed long-term residual vector [13].

Description

The function `ippsRPEQuantDecode_GSMFR` is declared in `ippsc.h` file. This function performs APCM inverse quantization of the input RPE samples. The output reconstructed long-term residual vector is formed as

$$pDst[i] = (2 \cdot pSrc[i] - 7) \cdot \frac{amp}{2^{amp1Sfs}}$$

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the `pSrc` or `pDst` pointer is NULL.

`ippStsScaleRangeErr` Indicates an error when `amp1Sfs` is less than 0.

Deemphasize_GSMFR

Performs de-emphasis filtering.

Syntax

```
IppStatus ippsDeemphasize_GSMFR_16s_I (Ipp16s* pSrcDst, int len,
Ipp16s* pMem);
```

Parameters

`pSrcDst` Pointer to the input short-term synthesized signal and output post-processed speech vector [`len`].

`len` Length of the input residual and output speech vectors.

`pMem` Pointer to the filter memory element.

Description

The function `ippsDeemphasize_GSMFR` is declared in `ippsc.h` file. This function performs de-emphasis of the input synthesized signal by filtering it through the filter with the following transfer function:

$$H(z) = \frac{1}{1 - \alpha z^{-1}}$$

where $\alpha = 0.86$ (28180 in Q15).

The initial memory of the filter will be set to zero.

The filtered speech signal is scaled up by multiple of 2 and then stored in *pSrcDst* with truncation of the three least significant bits.

Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when the <i>pSrcDst</i> or <i>pMem</i> pointer is NULL.
<i>ippStsRangeErr</i>	Indicates an error when <i>len</i> is less or equal to 0.

ShortTermAnalysisFilter_GSMFR

Performs short-term analysis filtering.

Syntax

```
IppStatus ippsShortTermAnalysisFilter_GSMFR_16s_I (const Ipp16s* pRC,
    Ipp16s* pSrcDstSpch, int len, Ipp16s* pMem);
```

Parameters

<i>pRC</i>	Pointer to the input reflection coefficients vector [8]: a_1, a_2, \dots, a_8 .
<i>pSrcDstSpch</i>	Pointer to the input pre-processed speech and output short term residual vector [<i>len</i>].
<i>len</i>	Length of the input speech and output residual vectors.
<i>pMem</i>	Pointer to the filter memory vector [8]: m_0, m_1, \dots, m_7 .

Description

The function `ippsShortTermAnalysisFilter_GSMFR` is declared in `ippsc.h` file. This function performs filtering of the input pre-processed speech vector $s(n)$ and stores the result in the output short-term residual vector $r(n)$ as given below:

$$r_0 = s(n)$$

$$r_i = r_{i-1} + a_i * m_{i-1}, \quad i = 1, \dots, 8$$

$$m_0 = s(n)$$

$$m_i = m_{i-1} + a_i * r_{i-1}, \quad i = 1, \dots, 7$$

$$r(n) = r_8$$

where $m_i, i = 0, \dots, 7$ is the filter memory and $r_i, i = 0, \dots, 8$ is the reusable local memory.

The initial filter memory vector will be zeroed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRC</code> , <code>pSrcDstSpch</code> or <code>pMem</code> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <code>len</code> is less or equal to 0.

ShortTermSynthesisFilter_GSMFR

Performs short-term synthesis filtering.

Syntax

```
IppStatus ippShortTermSynthesisFilter_GSMFR_16s (const Ipp16s* pRC, const
Ipp16s* pSrcResidual, Ipp16s* pDstSpch, int len, Ipp16s* pMem);
```

Parameters

<code>pRC</code>	Pointer to the input reflection coefficients vector [8]: a_1, a_2, \dots, a_8 .
<code>pSrcResidual</code>	Pointer to the input reconstructed short term residual vector [<code>len</code>].
<code>pDstSpch</code>	Pointer to the output speech vector [<code>len</code>].
<code>len</code>	Length of the input residual and output speech vectors.
<code>pMem</code>	Pointer to the filter memory vector [8]: m_0, m_1, \dots, m_7 .

Description

The function `ippsShortTermSynthesisFilter_GSMFR` is declared in `ippsc.h` file. This function performs filtering of the input reconstructed short-term residual $r(n)$ and stores the result in the output speech vector $s(n)$ as given below:

$$s_0 = r(n)$$

$$s_i = s_{i-1} - a_{9-i} * m_{8-i}, \quad i = 1, \dots, 8$$

$$m_{8-i} = m_{7-i} + -a_{9-i} * s_i, \quad i = 1, \dots, 7$$

$$s(n) = s_8$$

$$m_0 = s(n)$$

where $m_i, i = 0, \dots, 7$ is the filter memory and $s_i, i = 0, \dots, 8$ is the reusable local memory.

The initial filter memory vector will be zeroed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRC</code> , <code>pSrcResidual</code> , <code>pMem</code> , or <code>pDstSpch</code> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <code>len</code> is less or equal to 0.

HighPassFilter_GSMFR

Performs high-pass filtering of the input speech signal.

Syntax

```
IppStatus ippsHighPassFilter_GSMFR_16s (const Ipp16s* pSrc, Ipp16s* pDst,
    int len, int* pMem);
```

Parameters

<code>pSrc</code>	Pointer to the source speech vector [<code>len</code>].
<code>pDst</code>	Pointer to the destination filtered vector [<code>len</code>].

<i>len</i>	Length of the source and destination vectors.
<i>pMem</i>	Pointer to the filter memory vector [2].

Description

The function `ippsHighPassFilter_GSMFR` is declared in `ippsc.h` file. This function filters the input speech signal according to the transfer function:

$$H(z) = 0.5 \cdot \frac{1 - z^{-1}}{1 - \alpha z^{-1}}$$

where $\alpha = 0.99899$.

The initial filter memory will be set to zero.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pMem</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.

Schur_GSMFR

Estimates the reflection coefficients by Schur recursion.

Syntax

```
IppStatus ippsSchur_GSMFR_32s16s (const Ipp32s* pSrc, Ipp16s* pDst,
    int dstLen);
```

Parameters

<i>pSrc</i>	Pointer to the input autocorrelation vector [<i>dstLen</i> +1].
<i>pDst</i>	Pointer to the output reflection coefficients vector [<i>dstLen</i>].
<i>dstLen</i>	The number of reflection coefficients to estimate.

Description

The function `ippsSchur_GSMFR` is declared in `ippsc.h` file. This function implements the Schur algorithm according to GSM 06.10 clause 4.2.5. See also [ippsSchur](#) function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>dstLen</code> is less or equal to 0, or <code>dstLen</code> is less than 9.

WeightingFilter_GSMFR

Calculates the weighting filter.

Syntax

```
IppStatus ippsWeightingFilter_GSMFR_16s (const Ipp16s* pSrc, Ipp16s* pDst,
int dstLen);
```

Parameters

<code>pSrc</code>	Pointer to the input long-term residual vector <code>[-5,...,dstLen+4]</code> .
<code>pDst</code>	Pointer to the filtered output vector <code>[dstLen]</code> .
<code>dstLen</code>	The number of filtered elements to calculate.

Description

The function `ippsWeightingFilter_GSMFR` is declared in `ippsc.h` file. This function performs filtering of the input signal by symmetric FIR filter with predefined taps given below:

`taps[i]=[-134, -374, 0, 2054, 5741, 8192, 5741, 2054, 0, -374, -134]`, `i=0,...,10`

The filtering is performing according to formula:

$$dst[n] = \sum_{i=0}^{10} taps[i] \cdot src[n+i-5], \quad n = 0, ..., dstLen-1$$

The result of filtering is stored in *pDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>dstLen</i> is less or equal to 0.

Preemphasize_GSMFR

Computes pre-emphasis of a speech signal.

Syntax

```
IppStatus ippPreemphasize_GSMFR_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int* pMem, int len);
```

Parameters

<i>pSrc</i>	Pointer to the offset-free input speech signal.
<i>pDst</i>	Pointer to the pre-emphasized output signal.
<i>pMem</i>	Pointer to the filter memory value.
<i>len</i>	Length of the input and output signals.

Description

The function `ippPreemphasize_GSMFR` is declared in `ippsc.h` file. This function computes pre-emphasis of the input speech according to the difference signal pre-emphasis equation:

$$H(z) = 1 - \gamma z^{-1}$$

for $\gamma = -0.86$ and $pSrc[-1] = pMem[0]$.

The result of filtering is stored in *pDst*. The memory value *pMem*[0] is updated by *pSrc*[n-1]. For proper use of this function in GSM Full Rate codec, the memory value will be initialized to zero.

The function `ippPreemphasize_GSMFR` performs NR rounding (see [Rounding mode](#)).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pMem</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.

G.722.1 Related Functions

This chapter describes Intel IPP functions that can be used in implementing speech codecs following ITU-T recommendations G.722.1.
The list of these functions is given in the following table:

Table 9-7 Intel IPP G.722.1 Related Functions

Function Base Name	Operation
<code>DCTFwd_G722</code> , <code>DCTInv_G722</code>	Computes the forward or inverse discrete cosine transform (DCT) of a signal.
<code>DecomposeMLTToDCT</code>	Decomposes the MLT transform input signal to the form of the DCT input signal.
<code>DecomposeDCTToMLT</code>	Decomposes IDCT output signal to the form of the MLT transform output signal.
<code>HuffmanEncode_G722</code>	Performs Huffman encoding of the quantized amplitude envelope indexes.

These functions are used in the Intel IPP *G.722.1 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

DCTFwd_G722, DCTInv_G722

Computes the forward or inverse discrete cosine transform (DCT) of a signal.

Syntax

```
IppStatus ippSDCTFwd_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst);  
IppStatus ippSDCTInv_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst);
```

Parameters

<i>pSrc</i>	Pointer to the source vector[320].
<i>pDst</i>	Pointer to the destination vector[320]

Description

The functions `ippsDCTFwd_G722` and `ippsDCTInv_G722` are declared in `ippsc.h` file. These functions compute the forward and inverse discrete cosine transform (DCT) of length 320 as follows:

$$y(m) = \alpha \sum_{n=0}^{319} \cos\left(\frac{\pi}{320} \cdot (n+0.5) \cdot (m+0.5)\right) \cdot x(n) \quad , \text{ for forward DCT}$$

where $\alpha = 2/320$;

$$x(n) = \beta \sum_{m=0}^{319} \cos\left(\frac{\pi}{320} \cdot (m+0.5) \cdot (n+0.5)\right) \cdot y(m) \quad , \text{ for inverse DCT (IDCT)} .$$

where $\beta = 0.81$.

The formulae are the same except for the different scaling that is applied to meet the bit exact requirement.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>IppsStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is NULL.

DecomposeMLTToDCT

Decomposes the MLT transform input signal to the form of the DCT input signal.

Syntax

```
IppStatus ippsDecomposeMLTToDCT_G722_16s(const Ipp16s* pSrcSpch, Ipp16s*
pSrcDstSpchOld, Ipp16s* pDstSpchDecomposed);
```

Parameters

<code>pSrcSpch</code>	Pointer to the source vector [320].
-----------------------	-------------------------------------

pSrcSpchOld Pointer to the source/destination vector [320] of speech samples of the previous frame.

pDstSpchDecomposed Pointer to the destination vector [320].

Description

The function `ippSDecomposeMLTTToDCT_G722` is declared in `ippsc.h` file.

This function decomposes the input signal of the modulated lapped transform (MLT) to the form that fits DCT, so that the MLT transform may be performed in two steps: first, decomposition of the input signal, and second, DCT of the decomposed signal.

The decomposed speech signal is computed as follows:

$$v(n) = w(159 - n)x(159 - n) + w(160 + n)x(160 + n), 0 \leq n \leq 159$$

$$v(n + 160) = w(319 - n)(320 + n) - w(n)x(639 - n), 0 \leq n \leq 159$$

where

$$w(n) = \sin\left(\frac{\pi}{640}(n + 0.5)\right), 0 \leq n \leq 319$$

and

$$x(n) = \begin{cases} pSrcDstSpchOld[n], & 0 \leq n < 319 \\ pSrcSpch(n - 320), & 320 \leq n < 639 \end{cases}$$

The input signal *pSrcSpch* is stored in *pSrcDstSpchOld* for use in the next frame.

Return Values

`ippStsNoErr` Indicates no error.

`IppStsNullPtrErr` Indicates an error when the *pSrcSpch* or *pDstSpch* or *pSrcSpchOld* pointer is NULL.

DecomposeDCTToMLT

Decomposes IDCT output signal to the form of the MLT transform output signal.

Syntax

```
IppStatus ippsDecomposeDCTToMLT_G722_16s(const Ipp16s* pSrcSpchDecomposed,
    Ipp16s* pSrcDstSpchDecomposedOld, Ipp16s* pDstSpch);
```

Parameters

<i>pSrcSpchDecomposed</i>	Pointer to the source vector [320].
<i>pSrcSpchDecomposedOld</i>	Pointer to the source/destination vector [160] of decomposed speech samples of the previous frame.
<i>pDstSpch</i>	Pointer to the destination vector [320].

Description

The function `ippsDecomposeDCTToMLT_G722` is declared in `ippsc.h` file. This function decomposes the output signal of the Inverse DCT to the form that fits the output of the Inverse Modulated Lapped Transform (IMLT), so that the IMLT transform may be performed in two steps: first, Inverse Discrete Cosine Transform (IDCT), and second, decomposition of the IDCT output.

The decomposed signal is computed as follows:

$$pDstSpch(n) = w(n)u(159-n) + w(319-n)u_old(n), \quad 0 \leq n \leq 159$$

$$pDstSpch(n+160) = w(160+n)u(n) - w(159-n)u_old(159-n), \quad 0 \leq n \leq 159$$

where

$$w(n) = \sin\left(\frac{\pi}{640} \cdot (n + 0.5)\right), \quad 0 \leq n \leq 319$$

$u(n) = pSrcSpchDecomposed[n]$, and
 $u_old(n) = pSrcDstDecomposedOld[n]$.

The unused high half of the input signal is stored as $u_old()$ for use in the next frame:

$$u_old(n) = u(160+n), \quad 0 \leq n \leq 159$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pSrcSpch</i> or <i>pDstSpch</i> or <i>pSrcSpchOld</i> pointer is NULL.

HuffmanEncode_G722

Performs Huffman encoding of the quantized amplitude envelope indexes.

Syntax

```
IppStatus ippHuffmanEncode_G722_16s32u(int category, int qntAmpEnvIndex, const
    Ipp16s* pSrcMLTCoeffs, Ipp32u* pDstCode, int* pCodeLength);
```

Parameters

<i>category</i>	The category of the Modulated Lapped Transform (MLT) region in the range of [0-7].
<i>qntAmpEnvIndex</i>	The quantized amplitude envelope index in the range of [0-63].
<i>pSrcMLTCoeffs</i>	Pointer to the source vector [20] of raw MLT coefficients.
<i>pDstCode</i>	Pointer to the output Huffman code.
<i>pCodeLength</i>	Pointer to the output Huffman code length in bits.

Description

The function `ippHuffmanEncode_G722` is declared in `ippsc.h` file. This function performs Huffman encoding of the quantized index of the amplitude envelope of one of the twenty regions of the MLT coefficients.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pSrcMLTCoeffs</i> or <i>pDstCode</i> or <i>pCodeLength</i> pointer is NULL.

IppStsScaleRangeErrIndicates an error when *category* or *qntAmpEnvIndex* is out of proper range.

G.726 Related Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following ITU- T recommendations G.726 with Annex A.

These functions are used in the Intel IPP *G.726 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

The list of these functions is given in the following table:

Table 9-8 Intel IPP G.726.1 Related Functions

Function Base Name	Operation
EncodeGetStateSize_G726	Informative function, returns the number of bytes needed for encoder memory.
EncodeInit_G726	Initializes the memory for the ADPCM encoder.
Encode_G726	Performs ADPCM compression of the uniform PCM input.
DecodeGetStateSize_G726	Informative function, returns the number of bytes needed for decoder memory.
DecodeInit_G726	Initializes the memory for the G726 decoder.
Decode_G726	Performs decompression of the ADPCM bit-stream.

EncodeGetStateSize_G726

Informative function, returns the number of bytes needed for encoder memory.

Syntax

```
IppStatus ippsEncodeGetStateSize_G726_16s8u (unsigned int* pEncSize);
```

Parameters

pEncSize Pointer to the output memory size in bytes.

Description

The function `ippEncodeGetStateSize_G726` is declared in `ippsc.h` file. This function gets information about the amount of memory needed to process the G.726 ADPCM compression.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pEncSize</i> pointer is NULL.

EncodeInit_G726

Initializes the memory for the ADPCM encoder.

Syntax

```
IppStatus ippEncodeInit_G726_16s8u (IppsEncoderState_G726_16s* pEncMem,
    IppSpchBitRate rate);
```

Parameters

<i>pEncMem</i>	Pointer to the input memory buffer of size needed to properly initialize the G.726 encoder.
<i>rate</i>	Encode bit rate of the G.726 encoder, must be one of IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000, or IPP_SPCHBR_40000.

Description

The function `ippEncodeInit_G726` is declared in `ippsc.h` file. This function initializes the memory given by the pointer *pEncMem* to enable G.726 ADPCM compression starting from the reset state.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>rate</i> is not equal to one of the admissible encoding bit rates: IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000, or IPP_SPCHBR_40000.

`IppStsNullPtrErr` Indicates an error when the `pEncMem` pointer is NULL.

Encode_G726

Performs ADPCM compression of the uniform PCM input.

Syntax

```
IppStatus ippsEncode_G726_16s8u (IppsEncoderState_G726_16s* pEncMem,
    const Ipp16s* pSrc, Ipp8u* pDst, unsigned int len);
```

Parameters

<code>pEncMem</code>	Pointer to the memory buffer that has been initialized for ADPCM encode.
<code>pSrc</code>	Pointer to the uniform PCM input speech vector.
<code>pDst</code>	Pointer to the ADPCM bit-stream output vector.
<code>len</code>	The length of input/output vectors.

Description

The function `ippsEncode_G726` is declared in `ippsc.h` file. This function performs ADPCM compression of the 14-bit uniform PCM speech input (Recommendation G.726, Annex A) with the bit rate on which the G.726 encoder (with memory pointed to by `pEncMem`) was initialized to operate. Each byte of the output vector contains ADPCM compressed value of two, three, four, or five binary digits for 16, 24, 32 or 40 Kbit/s bit-rate ADPCM compression, respectively.

The Mu-Law or A-Law PCM input should be expanded to 14-bit uniform PCM prior to ADPCM compression (see Recommendation G.726). This expansion may be done, for example, by first applying the functions [ippsMuLawToLin_8u16s](#) or [ippsALawToMuLaw_8u16s](#) which expand 8-bit Mu-Law or A-Law PCM, respectively, into linear 16-bit PCM.

The linear 16-bit PCM input must be shifted two bits to the right (divided by four) to achieve the 14-bit uniform PCM input appropriate for the `ippsEncode_G726` function.

Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsBadArgErr</code>	Indicates an error when the <code>len</code> is less or equal to 0.
<code>IppStsNullPtrErr</code>	Indicates an error when the <code>pEncMem</code> , <code>pSrc</code> , or <code>pDst</code> pointer is NULL.

DecodeGetStateSize_G726

Informative function, returns the number of bytes needed for decoder memory.

Syntax

```
IppStatus ippDecodeGetStateSize_G726_8u16s (unsigned int* pDecSize);
```

Parameters

`pDecSize` Pointer to the output memory size in bytes.

Description

The function `ippDecodeGetStateSize_G726` is declared in `ippsc.h` file. This function reports the amount of memory needed to process the G.726 ADPCM decompression.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <code>pDecSize</code> pointer is NULL.

DecodeInit_G726

Initializes the memory for the G726 decoder.

Syntax

```
IppStatus ippDecodeInit_G726_8u16s (IppsDecoderState_G726_16s* pDecMem,  
    IppSpchBitRate rate, IppPCMLaw law);
```

Parameters

<i>pDecMem</i>	Pointer to the input memory buffer of size needed to properly initialize the G.726 decoder.
<i>rate</i>	Decode bit rate of the G.726 decoder, must be one of IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000, or IPP_SPCHBR_40000.
<i>law</i>	Output speech PCM law: must be one of IPP_PCM_MULAW, IPP_PCM_ALAW, or IPP_PCM_LINEAR.

Description

The function `ippsDecodeInit_G726` is declared in `ippsc.h` file. This function initializes the memory given by the pointer *pDecMem* to enable ADPCM decompression starting from the reset state.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>rate</i> is not equal to one of the admissible decoding bit rates: IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000, or IPP_SPCHBR_40000; or <i>law</i> is not equal to one of the acceptable output PCM values IPP_PCM_MULAW, IPP_PCM_ALAW, or IPP_PCM_LINEAR.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pDecMem</i> pointer is NULL.

Decode_G726

Performs decompression of the ADPCM bit-stream.

Syntax

```
IppStatus ippsDecode_G726_8u16s (IppsDecoderState_G726_16s* pDecMem, const
    Ipp8u* pSrc, Ipp16s* pDst, unsigned int len)
```

Parameters

<i>pDecMem</i>	Pointer to the memory buffer that has been initialized for G.726 ADPCM decoder.
----------------	---

<i>pSrc</i>	Pointer to the input vector that contains two, three, four, or five binary digits per byte for 16, 24, 32, or 40 Kbit/s ADPCM bit-stream, respectively.
<i>pDst</i>	Pointer to the 16-bit linear PCM speech output vector.
<i>len</i>	The length of input/output vectors.

Description

The function `ippsDecode_G726` is declared in `ippsc.h` file. This function performs decompression of the ADPCM bit-stream input (Recommendation G.726) into 16-bit linear PCM. The input bit-stream must be ADPCM compressed on the bit rate for which the G.726 decoder was initialized to operate.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pDecMem</i> , <i>pSrc</i> , or <i>pDst</i> pointer is NULL.

G.728 Related Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following ITU-T* recommendation G.728 with Annexes I, H.

These functions are used in the Intel IPP *G.728 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

The list of these functions is given in the following table:

Table 9-9 Intel IPP G.728 Related Functions

Function Base Name	Operation
<u>IIRGetStateSize_G728</u>	Gets the size of IIR state structure to be used.
<u>IIR_Init_G728</u>	Initializes the IIR state structure.
<u>IIR_G728</u>	Applies IIR filter to multiple samples.
<u>SynthesisFilterGetStateSize_G728</u>	Gets the size of synthesis filter state structure.
<u>SynthesisFilterInit_G728</u>	Initializes the synthesis filter state structure.
<u>SyntesisFilter_G728</u>	Applies the synthesis filter to multiple samples.
<u>CombinedFilterGetStateSize_G728</u>	Gets the size of combined filter state structure.
<u>CombinedFilterInit_G728</u>	Initializes the combined filter state structure.
<u>CombinedFilter_G728</u>	Applies the combined filter to multiple samples.
<u>PostFilterGetStateSize_G728</u>	Gets the size of the post filter state structure.
<u>PostFilterInit_G728</u>	Initializes the post filter state structure.
<u>PostFilter_G728</u>	Applies the post filter to multiple samples.
<u>PostFilterAdapterGetStateSize_G728</u>	Gets the size of the IppsPostFilterAdapterState structure to be used.
<u>PostFilterAdapterStateInit_G728</u>	Initializes the IppsPostFilterAdapterState structure.
<u>LPCInverseFilter_G728</u>	Computes the LPC prediction residual.
<u>PitchPeriodExtraction_G728</u>	Extracts pitch period from the LPC prediction residual.
<u>WinHybridGetStateSize_G728</u>	Gets the size of hybrid windowing module state structure.
<u>WinHybridInit_G728</u>	Initializes the hybrid windowing module state structure.
<u>WinHybrid_G728</u>	Applies the hybrid windowing.
<u>LevinsonDurbin_G728</u>	Calculates LP coefficients from the autocorrelation coefficients.
<u>CodebookSearch_G728</u>	Searches the codebook for the best code vector.
<u>ImpulseResponseEnergy_G728</u>	Implements shape codebook vector convolution and energy calculation.

IIRGetStateSize_G728

Gets the size of IIR state structure to be used.

Syntax

```
IppStatus ippsIIR16sGetStateSize_G728_16s (int* pSize);
```

Parameters

<i>pSize</i>	Pointer to the output IIR state size value.
--------------	---

Description

The function `ippsIIR16sGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the IIR filter.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.

IIR_Init_G728

Initializes the IIR state structure.

Syntax

```
IppStatus ippsIIR16sInit_G728_16s (IppsIIRState16s_G728_16s* pMem );
```

Parameters

<i>pMem</i>	Pointer to the memory allocated for IIR filter.
-------------	---

Description

The function `ippsIIR16sInit_G728` is declared in `ippsc.h` file. This function initializes the IIR state structure using the given memory block.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMem</i> pointer is NULL.

IIR_G728

Applies IIR filter to multiple samples.

Syntax

```
IppStatus ippsIIR16s_G728_16s (const Ipp16s* pCoeffs,
    const Ipp16s* pSrcQntSpeech, Ipp16s* pDstWgtSpeech, int len,
    IppsIIRState16s_G728_16s* pMem );
```

Parameters

<i>pCoeffs</i>	Pointer to the filter coefficients vector [20]: $b_0, \dots, b_9, a_0, \dots, a_9$ (in Q14).
<i>pSrcQntSpeech</i>	Pointer to the source vector [<i>len</i>].
<i>pDstWgtSpeech</i>	Pointer to the destination vector [<i>len</i>].
<i>len</i>	The number of source and destination samples.
<i>pMem</i>	Pointer to the IIR filter state structure.

Description

The function `ippsIIRState16s_G728` is declared in `ippsc.h` file. This function calculates the synthesized speech output by filtering the input quantized speech through the IIR filter one at a time according to the transfer function:

$$\frac{1 + \sum_{i=0}^9 b_i \cdot z^{-i}}{1 + \sum_{i=0}^9 a_i \cdot z^{-i}}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the <i>pCoeffs</i> , <i>pSrcQntSpeech</i> , <i>pDstWgtSpeech</i> or <i>pMem</i> pointers is NULL.

SynthesisFilterGetStateSize_G728

Gets the size of synthesis filter state structure.

Syntax

```
IppStatus ippSynthesisFilterGetStateSize_G728_16s (int* pSize);
```

Parameters

<i>pSize</i>	Pointer to the output size value of the synthesis filter state structure.
--------------	---

Description

The function `ippSynthesisFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the synthesis filter.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.

SynthesisFilterInit_G728

Initializes the synthesis filter state structure.

Syntax

```
IppStatus ippSynthesisFilterInit_G728_16s (IppsSynthesisFilterState_G728_16s*  
pMem);
```

Parameters

pMem Pointer to the memory allocated for synthesis filter.

Description

The function `ippsSynthesisFilterInit_G728` is declared in `ippsc.h` file. This function initializes synthesis filter state structure using the given memory block.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pMem* pointer is NULL.

SyntesisFilter_G728

Applies the synthesis filter to multiple samples.

Syntax

```
IppStatus ippsSyntesisFilterZeroInput_G728_16s (const Ipp16s* pCoeffs, Ipp16s*
pSrcDstExc, Ipp16s excSfs, Ipp16s* pDstSpeech, Ipp16s* pSpeechSfs,
IppsSynthesisFilterState_G728_16s* pMem);
```

Parameters

pCoeffs Pointer to the filter coefficients vector [51]: a_0, \dots, a_{50} in Q14.

pSrcDstExc Pointer to the input/output gain-scaled excitation vector [5].

excSfs The input scale of the previous gain-scaled excitation vector.

pDstSpeech Pointer to the output quantized speech vector [5].

pSpeechSfs The output scale of the quantized speech vector.

pMem Pointer to the synthesis filter state structure.

Description

The function `ippSyntesisFilterZeroInput_G728` is declared in `ippsc.h` file. This function computes the decoded speech vector as the sum of the zero-input response and the zero-state response of the synthesis filter according to the transfer function:

$$\frac{1}{1 + \sum_{i=1}^{50} a_i \cdot z^{-i}}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the <code>pCoeffs</code> , <code>pSrcDstExc</code> , <code>pDstSpeech</code> , <code>pSpeechSfs</code> or <code>pMem</code> pointer is NULL.

CombinedFilterGetStateSize_G728

Gets the size of combined filter state structure.

Syntax

```
IppStatus ippCombinedFilterGetStateSize_G728_16s (int* pSize);
```

Parameters

<code>pSize</code>	Pointer to the output size value of the combined filter state structure.
--------------------	--

Description

The function `ippCombinedFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the combined filter.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSize</code> pointer is NULL.

CombinedFilterInit_G728

Initializes the combined filter state structure.

Syntax

```
IppStatus ippsCombinedFilterInit_G728_16s (IppsCombinedFilterState_G728_16s*
    pMem);
```

Parameters

pMem Pointer to the memory allocated for the combined filter.

Description

The function `ippsCombinedFilterInit_G728` is declared in `ippsc.h` file. This function initializes combined filter state structure using the given memory block.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the *pMem* pointer is NULL.

CombinedFilter_G728

Applies the combined filter to multiple samples.

Syntax

```
IppStatus ippsCombinedFilterZeroInput_G728_16s(const Ipp16s* pSyntCoeff, const
    Ipp16s* pWgtCoeff, Ipp16s* pDstWgtZIR, IppsCombinedFilterState_G728_16s*
    pMem);
```

```
IppStatus ippsCombinedFilterZeroState_G728_16s(const Ipp16s* pSyntCoeff, const
    Ipp16s* pWgtCoeff, Ipp16s* pSrcDstExc, Ipp16s excSfs, Ipp16s* pDstSpeech,
    Ipp16s* pSpeechSfs, IppsCombinedFilterState_G728_16s* pMem);
```

Parameters

<i>pSyntCoeff</i>	Pointer to the filter coefficients vector [50]: a_1, \dots, a_{50} in Q14.
<i>pWgtCoeff</i>	Pointer to the filter coefficients vector [20]: $B_1, \dots, B_{10}, A_1, \dots, A_{10}$ in Q14.
<i>pSrcDstExc</i>	Pointer to the output gain-scaled excitation vector [5].
<i>excSfs</i>	The input scale of the previous gain-scaled excitation vector.
<i>pDstWgtZIR</i>	Pointer to the output zero input response vector [5] of the combined filter.
<i>pDstSpeech</i>	Pointer to the output quantized speech vector [5].
<i>pSpeechSfs</i>	The output scale of the quantized speech vector.
<i>pMem</i>	Pointer to the combined filter state structure.

Description

The functions `ippsCombinedFilterZeroInput_G728` and `ippsCombinedFilterZeroState_G728` are declared in `ippsc.h` file.

ippsCombinedFilterZeroInput_G728_16s. This function calculates the zero-input response of the combined filter by superposing two filters, specifically, the 50s-order synthesis filter and the 10s-order IIR filter according to the transfer function

$$\frac{1}{1 + \sum_{i=1}^{50} a_i \cdot z^{-i}} \cdot \frac{1 + \sum_{i=1}^{10} B_i \cdot z^{-i}}{1 + \sum_{i=1}^{10} A_i \cdot z^{-i}}$$

ippsCombinedFilterZeroState_G728_16s. This function first performs filtering of the gain-scaled excitation vector through the zero-state combined filter (see the above function). The memory of combined filter is then updated by adding zero-state responses of the synthesis and the IIR filters which it is combined of. The quantized speech output vector is obtained as a by-product of the memory updates.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSyntCoeff</i> , <i>pWgtCoeff</i> , <i>pSrcDstExc</i> , <i>pDstWgtZIR</i> , <i>pDstSpeech</i> , <i>pSpeechSfs</i> or <i>pMem</i> pointer is NULL.
-------------------------------	--

PostFilterGetStateSize_G728

Gets the size of the post filter state structure.

Syntax

```
IppStatus ippPostFilterGetStateSize_G728_16s (int* pSize);
```

Parameters

<i>pSize</i>	Pointer to the output size value of the post filter state structure.
--------------	--

Description

The function `ippPostFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the post filter.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.

PostFilterInit_G728

Initializes the post filter state structure.

Syntax

```
IppStatus ippPostFilterInit_G728_16s (IppsPostFilterState_G728_16s* pMem);
```

Parameters

<i>pMem</i>	Pointer to the memory allocated for the post filter.
-------------	--

Description

The function `ippsPostFilterInit_G728` is declared in `ippsc.h` file. This function initializes the post filter state structure using the given memory block.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMem</i> pointer is NULL.

PostFilter_G728

Applies the post filter to multiple samples.

Syntax

```
IppStatus ippsPostFilter_G728_16s (Ipp16s gl, Ipp16s glb, Ipp16s kp, Ipp16s
    tiltz, const Ipp16s* pCoeffs, const Ipp16s* pSrc, Ipp16s* pDst,
    IppsPostFilterState_G728_16s* pMem);
```

Parameters

<i>gl</i>	The LTP scaling factor.
<i>glb</i>	The LTP product term.
<i>kp</i>	The LTP lag, pitch period of the current frame.
<i>tiltz</i>	The STP tilt-compensation coefficient.
<i>pCoeffs</i>	Pointer to the post filter coefficients vector [20]: $B_1, \dots, B_{10}, A_1, \dots, A_{10}$.
<i>pSrc</i>	Pointer to the input speech vector [5]; elements $-kp, \dots, -1$ must be given as memory of the LTP filter.
<i>pDst</i>	Pointer to the output post-filtered speech vector [5].
<i>pMem</i>	Pointer to the post filter state structure.

Description

The function `ippsPostFilter_G728` is declared in `ippsc.h` file. This function performs filtering of input samples by one at a time according to the transfer function that is comprised of LTP filter and STP filter parts:

$$g_l \cdot (1 - g_{lb} \cdot z^{-10}) \cdot \frac{1 - \sum_{i=1}^{10} B_i \cdot z^{-i}}{1 - \sum_{i=1}^{10} A_i \cdot z^{-i}} \cdot (1 + tiltz \cdot z^{-1})$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pCoeffs</i> , <i>pSrc</i> , <i>pDst</i> or <i>pMem</i> pointer is NULL.

PostFilterAdapterGetStateSize_G728

Gets the size of the `IppsPostFilterAdapterState` structure to be used.

Syntax

```
IppStatus ippsPostFilterAdapterGetStateSize_G728 (int* pSize);
```

Parameters

<i>pSize</i>	Pointer to the output <code>IppsPostFilterAdapterState</code> structure size value.
--------------	---

Description

The function `ippsPostFilterAdapterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the postfilter adapter.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.

PostFilterAdapterStateInit_G728

Initializes the `IppsPostFilterAdapterState` structure.

Syntax

```
IppStatus ippPostFilterAdapterStateInit_G728 (
    IppsPostFilterAdapterState* pMem);
```

Parameters

<i>pMem</i>	Pointer to the memory allocated for postfilter adapter.
-------------	---

Description

The function `ippPostFilterAdapterStateInit_G728` is declared in `ippsc.h` file. This function initializes the `IppsPostFilterAdapterState` structure using the memory block given by *pMem*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMem</i> pointer is NULL.

LPCInverseFilter_G728

Computes the LPC prediction residual.

Syntax

```
IppStatus ippsLPCInverseFilter_G728_16s (const Ipp16s* pSrcSpeech, const
    Ipp16s* pCoeffs, Ipp16s* pDstResidual, IppsPostFilterAdapterState_G728*
    pMem) ;
```

Parameters

<i>pSrcSpeech</i>	Pointer to the quantized speech buffer [-240...0...4].
<i>pCoeffs</i>	Pointer to the 10 th -order LPC filter coefficients [10].
<i>pDstResidual</i>	Pointer to the LPC prediction residual vector [-140...0...99].
<i>pMem</i>	Pointer to the postfilter adapter memory.

Description

The function `ippsLPCInverseFilter_G728_16s` is declared in `ippsc.h` file. This function computes the LPC prediction residual vector for the current decoded speech vector. The 10th-order LPC inverse filter implements the following function:

$$A(z) = 1 - \sum_{i=0}^9 a_i \cdot z^{-(i+1)}$$

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

PitchPeriodExtraction_G728

Extracts pitch period from the LPC prediction residual.

```
IppStatus ippsPitchPeriodExtraction_G728_16s (const Ipp16s* pSrcResidual,
        int* pPitchPeriod, IppsPostFilterAdapterState_G728* pMem);
```

Parameters

<i>pSrcResidual</i>	Pointer to the LPC undecimated residual vector $d[-140,99]$.
<i>pPitchPeriod</i>	Pointer to the pitch period of the previous frame \bar{p} .
<i>pMem</i>	Pointer to the postfilter adapter memory.

Description

The function `ippsPitchPeriodExtraction_G728_16s` is declared in `ippsc.h` file. The function extracts the pitch period once per frame. It assumes that the input vector contains the LPC prediction residual of previous frames (samples $[-139,80]$) and of four vectors of the current frame stored, respectively, in elements $[80,84]$, $[85,89]$, $[90,94]$, and $[95,99]$. The prediction history for the previous frame is stored in the elements $[-139,80]$.

Calculation is as follows:

1. The last 20 elements of the residual computed for the current frame $[80,99]$ are filtered by the 3rd-order elliptic filter and then down-sampled by a factor of 4. The resulting five decimated LPC residual samples are added to the end of decimated LPC residual buffer $\bar{d}[\cdot]$ by shifting other 55 decimated LPC residual samples that were calculated for previous frames.
2. The time lag p_0 is identified which gives the maximum among seven autocorrelations of the undecimated LPC residual

$$\sum_{i=0}^{99} d[i] \cdot d[i - p], \quad p = 4\tau - 3, 4\tau - 2, \dots, 4\tau + 3$$

where the time lag τ is defined as a position where the maximum of the autocorrelations for the decimated LPC residual is held:

$$\sum_{i=0}^{24} \bar{d}[i] \cdot \bar{d}[i - \tau], \tau = 5, \dots, 35$$

3. The time lag p_1 is identified that gives the maximum of thirteen autocorrelations of the undecimated LPC residual around the pitch period of the previous frame \bar{p} :

$$\sum_{i=0}^{99} d[i] \cdot d[i - p], p = \bar{p} - 6, \dots, \bar{p} + 6$$

The two lags p_0, p_1 are checked to determine which of them should be set as a fundamental pitch period. First, the values β_0 and β_1 are calculated as

$$\beta_0 = \frac{\sum_{i=0}^{99} d[i] \cdot d[i - p_0]}{\sum_{i=0}^{99} d[i - p_0] \cdot d[i - p_0]}$$

$$\beta_1 = \frac{\sum_{i=0}^{99} d[i] \cdot d[i - p_1]}{\sum_{i=0}^{99} d[i - p_1] \cdot d[i - p_1]}$$

Then the output pitch period is chosen as follows:

$$pitchPeriod = \begin{cases} p_0 & \text{if } \beta_1 \leq 0.4 \cdot \beta_0 \\ p_1 & \text{if } \beta_1 > 0.4 \cdot \beta_0 \end{cases}$$

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

WinHybridGetStateSize_G728

Gets the size of hybrid windowing module state structure.

Syntax

```
IppStatus ippWinHybridGetStateSize_G728_16s (int M, int L, int N, int DIM, int* pSize);
```

Parameters

<i>M</i>	The input length of the LPC window.
<i>L</i>	The input adaptation cycle size in samples.
<i>N</i>	The input number of non-recursive window samples.
<i>DIM</i>	The input block size used for block scaling of the input speech by the <code>ippWinHybrid_G728</code> function.
<i>pSize</i>	Pointer to the output size value of the hybrid windowing module state structure.

Description

The function `ippWinHybridGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the hybrid windowing module according to the given window parameters.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>L</i> , <i>M</i> or <i>N</i> are less than or equal to zero.

WinHybridInit_G728

Initializes the hybrid windowing module state structure.

Syntax

```
IppStatus ippsWinHybridInit_G728_16s (const Ipp16s* pWinTab, int M, int L,
    int N, int DIM, Ipp16s a2L, IppsWinHybridState_G728_16s* pMem);
```

Parameters

<i>pWinTab</i>	The input vector of windowing coefficients.
<i>M</i>	The input length of LPC window. Must be not less than 10.
<i>L</i>	The input adaptation cycle size in samples.
<i>N</i>	The input number of non-recursive window samples.
<i>a2L</i>	The a^{2L} multiple used in calculation of the recursive component in hybrid windowing module.
<i>N</i>	The input number of non-recursive window samples.

Description

The function `ippsWinHybridInit_G728` is declared in `ippsc.h` file. This function initializes the hybrid windowing module state structure using the given memory block. The recursive component and the previous speech samples are zeroed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMem</i> pointer is NULL.

WinHybrid_G728

Applies the hybrid windowing.

Syntax

```
IppStatus ippsWinHybridBlock_G728_16s(Ipp16s bfi, const Ipp16s* pSrc, const
    Ipp16s* pSrcSfs, Ipp16s* pDst, IppsWinHybridState_G728_16s* pMem);
IppStatus ippsWinHybrid_G728_16s(Ipp16s bfi, const Ipp16s* pSrc, const
    Ipp16s* pSrcSfs, Ipp16s* pDst, IppsWinHybridState_G728_16s* pMem);
```

Parameters

<i>bfi</i>	The input bad frame indicator: “1” signifies a bad frame, any other value signifies a good frame.
<i>pSrc</i>	Pointer to the input speech vector [20].
<i>pSrcSfs</i>	Pointer to the input vector [5] of scale factors for elements of the input speech vector. Each element of the <i>pSrcSfs</i> vector may specify a separate scale factor for a block of input speech vector elements. Thus, <i>pSrcSfs</i> [0] is the scale factor of the elements <i>pSrc</i> [0].. <i>pSrc</i> [<i>DIM</i> -1]; <i>pSrcSfs</i> [1] is the scale factor for <i>pSrc</i> [<i>DIM</i>].. <i>pSrc</i> [2* <i>DIM</i> -1], and so on, where <i>DIM</i> is a block length that must be defined by the <i>WinHybridInit_G728()</i> function.
<i>pDst</i>	Pointer to the output reflection coefficients vector [M+1], where M is the length of the LPC window defined in <i>WinHybridInit_G728()</i> function.
<i>pMem</i>	Pointer to the post filter state structure.

Description

The function *ippsWinHybrid_G728* is declared in *ippsc.h* file. This function first applies the window to the input speech vector and then calculates the autocorrelation coefficients needed in LPC analysis by the formulae:

$$R_m(i) = r_m(i) + \sum_{k=-M}^{m-1} s_m(k) s_m(k-i), m = 0, \dots, M$$

where the recursive component of adaptation cycle is calculated as follows:

$$r_m(i) = \alpha^{2L} \cdot r_{m-L}(i) + \sum_{k=m-L-N}^{m-N-1} s_m(k) s_m(k-i)$$

The recursive part is calculated using the data calculated in previous adaptation cycle and stored in module memory.

A *white noise correction* is applied to autocorrelation coefficients by increasing the energy as follows:

$$r_m(0) = \frac{257}{256} \cdot r_m(0)$$

The recursive component and previous speech samples are stored in the module memory and may be used in the next adaptation cycle.

If the bad frame indicator is on, then only 10 autocorrelation coefficients are calculated and output.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcSfs</i> , <i>pSrc</i> , <i>pDst</i> or <i>pMem</i> pointer is NULL.

LevinsonDurbin_G728

Calculates LP coefficients from the autocorrelation coefficients.

Syntax

```
IppStatus ippLevinsonDurbin_G728_16s_Sfs(const Ipp16s* pSrcAutoCorr, int
    order, Ipp16s* pDstLPC, Ipp16s* pDstResidualEnergy, Ipp16s*
    pDstScaleFactor);

IppStatus ippLevinsonDurbin_G728_16s_ISfs(const Ipp16s* pSrcAutoCorr, int
    numSrcLPC, int order, Ipp16s* pSrcDstLPC, Ipp16s* pSrcDstResidualEnergy,
    Ipp16s* pSrcDstScaleFactor);
```

Parameters

<i>pSrcAutoCorr</i>	Pointer to the input autocorrelation coefficients vector [<i>order</i> +1].
<i>order</i>	The input number of LP coefficients to calculate.
<i>numSrcLPC</i>	The input number of pre-calculated LPCs.
<i>pDstLPC</i>	Pointer to the output LPC vector [<i>order</i>].
<i>pSrcDstLPC</i>	Pointer to the input/output LPC vector [<i>order</i>].
<i>pDstResidualEnergy</i>	Pointer to the output residual energy.
<i>pSrcDstResidualEnergy</i>	Pointer to the input/output residual energy of the pre-calculated LPC.
<i>pDstScaleFactor</i>	Pointer to the output scale factor of the LPC vector.
<i>pSrcDstScaleFactor</i>	Pointer to the input scale factor of the pre-calculated LPC and the output scale factor of the output LPC vector.

Description

The functions `ippsLevinsonDurbin_G728_16s_Sfs` and `ippsLevinsonDurbin_G728_16s_ISfs` are declared in `ippsc.h` file.

ippsLevinsonDurbin_G728_16s_Sfs. This function may be used to calculate Linear Prediction (LP) coefficients by solving the following set of linear equations:

$$\sum_{i=0}^{order-1} a_i \cdot r(|i-k|) = r(k) \quad k = 1, 2, \dots, order$$

where a_i , $i = 0, 1, \dots, order-1$ are the LP coefficients to be calculated and stored in the output LPC vector. The description of the Levinson-Durbin algorithm used by this function may be found in [ippsLevinsonDurbin_G729](#) function.

Both `ippsLevinsonDurbin_G728` and `ippsLevinsonDurbin_G729B` calculate mathematically the same, but not bit exact LPCs. The difference is that the `ippsLevinsonDurbin_G729B` function outputs LPCs in Q12, while the `ippsLevinsonDurbin_G728` function automatically rescales the LPCs if overflow occurs.

ippsLevinsonDurbin_G728_16s_Isfs. This function may be used to continue Levinson-Durbin recursion for bigger *order*. The LPCs, their scale factor and the residual energy of the previous recursion and the additional autocorrelation coefficients are used to resume recursion.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>order</i> is less than or equal to 0, or when <i>order</i> < <i>numSrcLPC</i> for the in-place function.

CodebookSearch_G728

Searches the codebook for the best code vector.

Syntax

```
IppStatus ippsCodebookSearch_G728_16s(const Ipp16s* pSrcCorr, const Ipp16s*
    pSrcEnergy, int* pDstShapeIdx, int* pDstGainIdx, short* pDstCodebookIdx,
    IppSpchBitRate rate);
```

Parameters

<i>pSrcCorr</i>	Pointer to the input vector [5] of time-reversed convolution of the target signal.
<i>pSrcEnergy</i>	Pointer to the input vector [128] of the energy of convolved shape codevector.
<i>pDstShapeIdx</i>	Pointer to the output best 7-bit shape codebook index.
<i>pDstGainIdx</i>	Pointer to the output best 3-bit gain codebook index.
<i>pDstCodebookIdx</i>	Pointer to the output best codebook index to be transmitted.
<i>rate</i>	Input coding bit rate.

Description

The function `ippsCodebookSearch_G728` is declared in `ippsc.h` file. This function implements the “Error calculator and best codebook index selector” block used to search through the gain codebook and the shape codebook for the best combination of the gain and shape codebook indexes.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcCorr</code> , <code>pSrcEnergy</code> , <code>pDstShapeIdx</code> , <code>pDstGainIdx</code> or <code>pDstCodebookIdx</code> pointer is NULL.
<code>IppStsRangeErr</code>	Indicates an error when <code>rate</code> is not one of the acceptable values <code>IPP_SPCHBR_16000</code> , <code>IPP_SPCHBR_12800</code> or <code>IPP_SPCHBR_9600</code> for the 16, 12.8 or 9.6 Kbit/s coding bit rates, respectively.

ImpulseResponseEnergy_G728

Implements shape codebook vector convolution and energy calculation.

Syntax

```
IppStatus ippsImpulseResponseEnergy_G728_16s(const Ipp16s* pSrcImpResp, Ipp16s*
pDstEnergy);
```

Parameters

<code>pSrcImpResp</code>	Pointer to the input impulse response vector [5] of the synthesis and weighted filter.
<code>pDstEnergy</code>	Pointer to the output energy vector [128] of the convolved shape codevector.

Description

The function `ippsImpulseResponseEnergy_G728` is declared in `ippsc.h` file. This function implements the “Shape codevector convolution and energy calculator” block used to calculate the energy of the convolved shape codevector.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcImpResp</i> or <i>pDstEnergy</i> pointer is NULL.

Echo Canceller Related Functions

Intel IPP functions described in this section implement building blocks that can be used to create an acoustic and network echo cancellers compliant to the appropriate ITU-T Recommendations: G.168-2000, and G.167 (superseded by G.161, P.340).

The list of Intel IPP Echo Canceller related functions is given in the following table:

Table 9-10 Intel IPP Echo Canceller Related Functions

Function Base Name	Operation
<u>SubbandProcessGetSize</u>	Returns size of the subband process state structure.
<u>SubbandProcessInit</u>	Initializes the subband process state structure.
<u>SubbandAnalysis</u>	Decomposes a frame into a complex subband representation
<u>SubbandSynthesis</u>	Reconstructs frame from a complex subband representation.
<u>SubbandControllerGetSize_EC</u>	Returns size of the subband controller state structure.
<u>SubbandControllerInit_EC</u>	Initializes the subband controller state structure.
<u>SubbandControllerUpdate_EC</u>	Updates subband controller state and returns the step sizes.
<u>SubbandController_EC</u>	Updates filter coefficients and returns output gain coefficients.
<u>SubbandControllerReset_EC</u>	Resets subband controller state.
<u>ToneDetectGetStateSize_EC</u>	Returns size of the tone detector state structure.
<u>ToneDetectInit_EC</u>	Initializes the tone detector state structure.
<u>ToneDetect_EC</u>	Detects the signal of 2100 Hz frequency with every 450 ms phase reversal.
<u>FullbandControllerGetSize_EC</u>	Returns size of the fullband controller state structure.
<u>FullbandControllerInit_EC</u>	Initializes the fullband controller state structure.
<u>FullbandControllerUpdate_EC</u>	Updates the fullband controller state and returns the step sizes.

Table 9-10 Intel IPP Echo Cancellor Related Functions (continued)

Function Base Name	Operation
FullbandController_EC	Updates filter coefficients and returns output gain coefficients.
FullbandControllerReset_EC	Resets fullband controller state.
FIR_EC	Computes FIR filter results.
NLMS_EC	Performs FIR filtering with coefficients update.

SubbandProcessGetSize

Returns size of the subband process state structure.

Syntax

```
IppStatus ippsSubbandProcessGetSize_32f(int order, int windowLen, int*
    pStateSize, int* pInitBufSize, int* pBufSize);
IppStatus ippsSubbandProcessGetSize_16s(int order, int windowLen, int*
    pStateSize, int* pInitBufSize, int* pBufSize);
```

Parameters

<i>order</i>	Number of subbands is equal to $2^{order-1} + 1$.
<i>windowLen</i>	Length of window.
<i>pStateSize</i>	Pointer to the computed value of buffer size for the subband process state structure.
<i>pInitBufSize</i>	Pointer to the computed buffer size for use in the initialization function.
<i>pBufSize</i>	Pointer to the computed size of work buffer.

Description

The functions `ippsSubbandProcessGetSize` are declared in the `ippsc.h` file. These functions return the sizes of memory buffers required for the functions [ippsSubbandAnalysis](#) and [ippsSubbandSynthesis](#) to operate. The memory must be allocated and initialized by means of the [ippsSubbandProcessInit](#) function prior to using the functions `ippsSubbandAnalysis` or `ippsSubbandSynthesis`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pStateSize</i> , <i>pInitBufSize</i> , or <i>pBufSize</i> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>order</i> is less than or equal to 0 or when <i>windowLen</i> is not divisible by 2^{order} .

SubbandProcessInit

Initializes the subband process state structure.

Syntax

```
IppStatus ippsSubbandProcessInit_32f(IppsSubbandProcessState_32f* pState, int
    order, int frameSize, int windowLen, const Ipp32f* pWindow, Ipp8u*
    pInitBuf);
```

```
IppStatus ippsSubbandProcessInit_16s(IppsSubbandProcessState_16s* pState, int
    order, int frameSize, int windowLen, const Ipp32s* pWindow, Ipp8u*
    pInitBuf);
```

Parameters

<i>pState</i>	Pointer to the subband process state structure to be created.
<i>order</i>	Number of subbands is equal to $2^{order-1} + 1$.
<i>frameSize</i>	Size of frame. Must be in range from 1 to 2^{order} .
<i>windowLen</i>	Window length.
<i>pWindow</i>	Pointer to window coefficients . May be NULL if the <i>order</i> , <i>frameSize</i> and <i>windowLen</i> are equal to one of the predefined sets: (5, 24, 128) or (6, 44, 256). In this case the predefined window is used.
<i>pInitBuf</i>	Pointer to the initialized buffer.

Description

The functions `ippSubbandProcessInit` are declared in the `ippsc.h` file. These functions initialize the state structure and internal data of the subband process algorithm. The [ippSubbandAnalysis](#) and [ippSubbandSynthesis](#) functions use this memory in their operation.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pInitBuf</code> or <code>pState</code> is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <code>order</code> is less or equal to 0, or when <code>windowLen</code> is not divisible by 2^{order} , or when <code>frameSize</code> is less than or equal to 0, or when <code>pWindow</code> is <code>NULL</code> and the appropriate internal window does not exist.

SubbandAnalysis

Decomposes a frame into a complex subband representation.

Syntax

```
IppStatus ippSubbandAnalysis_32f32fc(const Ipp32f* pSignal, Ipp32fc*
    pSubbands, IppsSubbandProcessState_32f* pState, Ipp8u* pBuf);
IppStatus ippSubbandAnalysis_16s32sc_Sfs(const Ipp16s* pSignal, Ipp32sc*
    pSubbands, IppsSubbandProcessState_16s* pState, int scaleFactor, Ipp8u*
    pBuf);
```

Parameters

<code>pState</code>	Pointer to the subband process state structure.
<code>pSignal</code>	Pointer to the source vector. Vector length must be equal to the frame size which was used to initialize the subband process algorithm by ippSubbandProcessInit function.

<i>pSubbands</i>	Pointer to the resulting subband vector. Vector length is equal to the number of subbands calculated as $2^{order-1} + 1$, where <i>order</i> is specified in the function ippsSubbandProcessInit .
<i>pBuf</i>	Pointer to the work buffer of size specified by the parameter <i>pBufSize</i> in the ippsSubbandProcessGetSize function. This buffer must be allocated by the user.

Description

The functions `ippsSubbandAnalysis` are declared in the `ippsc.h` file. These functions perform subband analysis using the oversampled DFT filter bank.

Computation steps are as follows:

1) Calculate the weighted sum as

$$X(j) = \sum_{i=0}^{windowLen/fftLen-1} window(i \cdot fftLen + j) \cdot x(i \cdot fftLen + j)$$

for $j = 0, \dots, fftLen - 1$, $fftLen = 2^{order}$.

Here x denotes the input sample buffer of size *windowLen*.

2) Calculate and return the Fast Fourier Transform of X .

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.

SubbandSynthesis

Reconstructs frame from a complex subband representation.

Syntax

```
IppStatus ippsSubbandSynthesis_32fc32f(const Ipp32fc* pSubbands, Ipp32f*
    pSignal, IppsSubbandProcessState_32f* pState, Ipp8u* pBuf);
```

```
IppStatus ippsSubbandSynthesis_32sc16s_Sfs(const Ipp32sc* pSubbands, Ipp16s*
    pSignal, IppsSubbandProcessState_16s* pState, int scaleFactor, Ipp8u* pBuf);
```

Parameters

<i>pState</i>	Pointer to the subband process state structure.
<i>pSubbands</i>	Pointer to the resulting subband vector. Vector length is equal to the number of subbands calculated as $2^{order-1} + 1$, where <i>order</i> is specified in the function ippsSubbandProcessInit .
<i>pSignal</i>	Pointer to the destination signal vector. Vector length is equal to frame size which was used to initialize the subband process algorithm by ippsSubbandProcessInit function.
<i>pBuf</i>	Pointer to the work buffer of size computed by the function ippsSubbandProcessGetSize (returned via <i>pBufSize</i> parameter).

Description

The functions `ippsSubbandSynthesis` are declared in the `ippsc.h` file. These functions perform subband synthesis using the oversampled DFT filter bank.

Computation steps are as follows:

- 1) Calculates IX as the inverse Fast Fourier Transform of the input subband sample X . Transform length is $fftLen = 2^{order}$.
- 2) Shifts internal buffer by *frameSize* and sets to zero the undefined buffer values.
- 3) Updates buffer values as

$$x(j) := x(j) + window(j) * IX(j \% fftLen),$$
 for $j = 0, \dots, windowLen - 1$
- 4) Returns most recent *frameSize* samples from the buffer.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.

SubbandControllerGetSize_EC

Returns size of the subband controller state structure.

Syntax

```
IppStatus ippsSubbandControllerGetSize_EC_32f(int numSubbands, int frameSize,
      int numSegments, ippECFrequency sampleFreq, int* pSize);
IppStatus ippsSubbandControllerGetSize_EC_16s(int numSubbands, int frameSize,
      int numSegments, ippECFrequency sampleFreq, int* pSize);
```

Parameters

<i>numSubbands</i>	Number of subbands.
<i>frameSize</i>	Size of the frame.
<i>numSegments</i>	Number of segments.
<i>sampleFreq</i>	Sample frequency.
<i>pSize</i>	Pointer to the computed buffer size value.

Description

The functions `ippsSubbandControllerGetSize_EC` are declared in the `ippsc.h` file. These functions return the sizes of memory required for the [ippsSubbandControllerUpdate_EC](#) and [ippsSubbandController_EC](#) functions to operate. The memory must be allocated and initialized by means of the [ippsSubbandControllerInit_EC](#) function prior to using the functions `ippsSubbandControllerUpdate_EC` and `ippsSubbandController_EC`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>numSubbands</i> is less than or equal to 0, or <i>numSegment</i> is less than or equal to 0, or <i>frameSize</i> is less than or equal to 0.

SubbandControllerInit_EC

Initializes the subband controller state structure.

Syntax

```
IppStatus ippsSubbandControllerInit_EC_32f(IppsSubbandControllerState_EC_32f*
    pState, int numSubbands, int frameSize, int numSegments, ippECFrequency
    sampleFreq);

IppStatus ippsSubbandControllerInit_EC_16s(IppsSubbandControllerState_EC_16s*
    pState, int numSubbands, int frameSize, int numSegments, ippECFrequency
    sampleFreq);
```

Parameters

<i>pState</i>	Pointer to the subband controller state structure to be created.
<i>numSubbands</i>	Number of subbands.
<i>frameSize</i>	Size of the frame.
<i>numSegments</i>	Number of segments.
<i>sampleFreq</i>	Sample frequency.

Description

The functions `ippsSubbandControllerInit_EC` are declared in the `ippsc.h` file. These functions initialize the state structure and the internal data of the subband controller algorithm. The functions [ippsSubbandControllerUpdate_EC](#) and [ippsSubbandController_EC](#) use this memory in their operation.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>numSubbands</i> is less than or equal to 0, or <i>numSegments</i> is less than or equal to 0, or <i>frameSize</i> is less than or equal to 0.

SubbandControllerUpdate_EC

Updates controller state and returns the step sizes.

Syntax

```
IppStatus ippsSubbandControllerUpdate_EC_32f(const Ipp32f* pSrcRin, const
    Ipp32f* pSrcSin, const Ipp32fc** ppSrcRinSubbandsHistory, const Ipp32fc*
    pSrcSinSubbands, double* pDstStepSize, IppsSubbandControllerState_EC_32f*
    pState);

IppStatus ippsSubbandControllerUpdate_EC_16s(const Ipp16s* pSrcRin, const
    Ipp16s* pSrcSin, const Ipp32sc** ppSrcRinSubbandsHistory, const Ipp32sc*
    pSrcSinSubbands, IppAECscaled32s* pDstStepSize,
    IppsSubbandControllerState_EC_16s* pState);
```

Parameters

<i>pState</i>	Pointer to the subband controller state structure.
<i>pSrcRin</i>	Pointer to receive-in signal frame.
<i>pSrcSin</i>	Pointer to send-in signal frame. Frame size is specified in ippsSubbandControllerInit_EC function.
<i>ppSrcRinSubbandsHistory</i>	Pointer to an array of pointers to the most recent receive-in blocks. Size of the array is equal to <i>numSegments</i> specified in ippsSubbandControllerInit_EC function.
<i>pSrcSinSubbands</i>	Pointer to subband representation of send-in signal frame (or NULL). Size of the array is equal to <i>numSubbands</i> specified in ippsSubbandControllerInit_EC function.
<i>pDstStepSize</i>	Pointer to the vector of step sizes. Vector length is equal to <i>numSubbands</i> specified in ippsSubbandControllerInit_EC function.

Description

The functions `ippsSubbandControllerUpdate_EC` are declared in the `ippsc.h` file. These functions update the internal state of the subband controller and return the step size values for further use in the coefficients' adaptation process.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pState</i> , <i>pRin</i> , <i>pSin</i> , <i>pRinSubbands</i> , or <i>pStepSize</i> is NULL.

SubbandController_EC

The main subband controller function which updates filter coefficients and returns output gain coefficients.

Syntax

```
IppStatus ippsSubbandController_EC_32f(const Ipp32fc* pSrcAdaptiveFilterErr,
    const Ipp32fc* pSrcFixedFilterErr, Ipp32fc** ppDstAdaptiveCoefs, Ipp32fc**
    ppDstFixedCoefs, Ipp32f* pDstSGain, IppsSubbandControllerState_EC_32f*
    pState);

IppStatus ippsSubbandController_EC_16s(const Ipp32sc* pSrcAdaptiveFilterErr,
    const Ipp32sc* pSrcFixedFilterErr, Ipp32sc** pDstpAdaptiveCoefs, Ipp32sc**
    ppFixedCoefs, Ipp32s* pDstSGain, IppsSubbandControllerState_EC_16s* pState);
```

Parameters

<i>pState</i>	Pointer to SubbandController_EC state structure.
<i>pSrcAdaptiveFilterErr</i>	Pointer to the adaptive filter error vector. Vector length is equal to <i>numSubbands</i> , specified in ippsSubbandControllerInit_EC function.
<i>pSrcFixedFilterErr</i>	Pointer to the fixed filter error vector. Vector length is equal to <i>numSubbands</i> .
<i>ppDstAdaptiveCoefs</i>	Pointer to an array of pointers to the adaptive filter coefficients vectors. Size of the array is equal to <i>numSegments</i> specified in ippsSubbandControllerInit_EC function.
<i>ppDstFixedCoefs</i>	Pointer to an array of pointers to the fixed filter coefficients vectors. Size of the array is equal to <i>numSegments</i> specified in ippsSubbandControllerInit_EC function.
<i>pDstSGain</i>	Pointer to the send gain coefficient.

Description

The functions `ippsSubbandController_EC` are declared in the `ippsc.h` file.

These functions compare powers of adaptive filter error and fixed filter error and track changes of adaptive filter coefficients power.

If the adaptive filter has a significantly smaller error and is stable (which is indicated by coefficients power changing slowly), its coefficients are copied to the fixed filter. This case corresponds to “no double-talk” mode.

If the fixed filter has a significantly smaller error, its coefficients are copied to the adaptive filter. This case corresponds to the double-talk mode.

Send gain coefficient calculation (non-linear processor technology) is based on presence of the double-talk mode and changes of receive-in and send-in signal powers.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

SubbandControllerReset_EC

Resets the subband controller state.

Syntax

```
IppStatus ippsSubbandControllerReset_EC_32f(IppsSubbandControllerState_EC_32f*  
    pState);  
IppStatus ippsSubbandControllerReset_EC_16s(IppsSubbandControllerState_EC_16s*  
    pState);
```

Parameters

<i>pState</i>	Pointer to the subband controller state structure.
---------------	--

Description

The functions `ippsSubbandControllerReset_EC` are declared in the `ippsc.h` file. These functions reset the subband controller state. You can use these functions after an interruption in adaptation process occurred.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pState</code> pointer is NULL.

ToneDetectGetStateSize_EC

Returns size of the tone detector state structure.

Syntax

```
IppStatus ippsToneDetectGetStateSize_EC_16s(ippECFrequency sampleFreq, int*
    pSize);
IppStatus ippsToneDetectGetStateSize_EC_32f(ippECFrequency sampleFreq, int*
    pSize);
```

Parameters

<code>sampleFreq</code>	Sample frequency.
<code>pSize</code>	Pointer to the computed buffer size value.

Description

The functions `ippsToneDetectGetStateSize_EC` are declared in the `ippsc.h` file. These functions return the size of memory buffers required for the [ippsToneDetect_EC](#) functions to operate.

The appropriate memory must be allocated and initialized by means of the [ippsToneDetectInit_EC](#) function prior to using the `ippsToneDetect_EC` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSize</code> pointer is NULL.

<code>ippStsRangeErr</code>	Indicates an error when <i>sampleFreq</i> is not a valid element of the enumerated type <code>IppFrequency</code> .
-----------------------------	---

ToneDetectInit_EC

Initializes the tone detector state structure.

Syntax

```
IppStatus ippstToneDetectInit_EC_16s(IppsToneDetectState_EC_16s* pState,
    ippECFrequency sampleFreq);
IppStatus ippstToneDetectInit_EC_32f(IppsToneDetectState_EC_32f* pState,
    ippECFrequency sampleFreq);
```

Parameters

<i>pState</i>	Pointer to the memory buffer to be initialized as the tone detector state structure. The memory size must be computed by the ippstToneDetectGetStateSize_EC function and returned via <i>pSize</i> parameter.
<i>sampleFreq</i>	Sample frequency.

Description

The functions `ippstToneDetectInit_EC` are declared in the `ippsc.h` file. These functions initialize the internal data of the tone detect algorithm. The [ippstToneDetect_EC](#) functions use this memory in their operation.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsRangeErr</code>	Indicates an error when <i>sampleFreq</i> is not a valid element of the enumerated type <code>IppFrequency</code> .

ToneDetect_EC

Detects the signal of 2100 Hz frequency with every 450 ms phase reversal.

```
IppStatus ippstToneDetect_EC_16s(const Ipp16s* pSignal, int len, int* pResult,
    IppstToneDetectState_EC_16s* pState);
IppStatus ippstToneDetect_EC_32f(const Ipp32f* pSignal, int len, int* pResult,
    IppstToneDetectState_EC_32f* pState);
```

Parameters

<i>pState</i>	Pointer to the tone detector state structure.
<i>pSignal</i>	Pointer to signal vector.
<i>len</i>	Number of samples in signal vector.
<i>pResult</i>	Pointer to the result value. If the value is not zero, this means that the tone was detected.

Description

The functions `ippstToneDetect_EC` are declared in the `ippsc.h` file.

These functions apply an infinite impulse response (IIR) band-pass filter centered at 2100 Hz frequency. Functions compare powers of the signal before and after the filter.

High-level signal after the filter with periodical short blips is considered as the tone detected.



NOTE. Detection can occur when summary length of processed signal vectors is equal to or greater than 900 ms.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

FullbandControllerGetSize_EC

Returns size of the fullband controller state structure.

Syntax

```
IppStatus ippsFullbandControllerGetSize_EC_32f(int frameSize, int tapLen,
        ippECFrequency sampleFreq, int* pSize);
IppStatus ippsFullbandControllerGetSize_EC_16s(int frameSize, int tapLen,
        ippECFrequency sampleFreq, int* pSize);
```

Parameters

<i>frameSize</i>	Size of the frame.
<i>tapLen</i>	Number of tap values.
<i>sampleFreq</i>	Sample frequency.
<i>pSize</i>	Pointer to the computed buffer size value.

Description

The functions `ippsFullbandControllerGetSize_EC` are declared in the `ippsc.h` file. These functions return the sizes of memory buffers required for the functions [ippsFullbandControllerUpdate_EC](#) and [ippsFullbandController_EC](#) to operate. The appropriate memory must be allocated and initialized by calling the function [ippsFullbandControllerInit_EC](#) prior to using the `ippsFullbandControllerUpdate_EC`, `ippsFullbandControllerReset_EC` or `ippsFullbandController_EC` functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>frameSize</i> is less than or equal to 0, or <i>tapLen</i> is less than or equal to 0.

FullbandControllerInit_EC

Initializes the fullband controller state structure.

Syntax

```
IppStatus ippsFullbandControllerInit_EC_32f(IppsFullbandControllerState_EC_32f*
    pState, int frameSize, int tapLen, ippeCFrequency sampleFreq);
IppStatus ippsFullbandControllerInit_EC_16s(IppsFullbandControllerState_EC_16s*
    pState, int frameSize, int tapLen, ippeCFrequency sampleFreq);
```

Parameters

<i>pState</i>	Pointer to the memory buffer required for the fullband controller state structure to be initialized. The size of the memory buffer must be computed by ippsFullbandControllerGetSize_EC function and returned via <i>pSize</i> parameter.
<i>frameSize</i>	Size of the frame.
<i>tapLen</i>	Number of tap values.
<i>sampleFreq</i>	Sample frequency.

Description

The functions `ippsFullbandControllerInit_EC` are declared in the `ippsc.h` file. These functions initialize the internal data of the fullband controller algorithm. The [ippsFullbandControllerUpdate_EC](#) and [ippsFullbandController_EC](#) functions use this memory in their operation.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pState</i> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>frameSize</i> is less than or equal to 0, or <i>tapLen</i> is less than or equal to 0.

FullbandControllerUpdate_EC

Updates the fullband controller state and returns the step sizes.

Syntax

```
IppStatus ippsFullbandControllerUpdate_EC_32f(const Ipp32f* pSrcRin, const
    Ipp32f* pSrcSin, Ipp32f* pDstStepSize, IppsFullbandControllerState_EC_32f*
    pState);

IppStatus ippsFullbandControllerUpdate_EC_16s(const Ipp16s* pSrcRin, const
    Ipp16s* pSrcSin, IppAECScaled32s* pDstStepSize,
    IppsFullbandControllerState_EC_16s* pState);
```

Parameters

<i>pState</i>	Pointer to the fullband controller state structure.
<i>pSrcRin</i>	Pointer to the receive-in signal history. The history length is <i>tapLen</i> + <i>frameSize</i> , where <i>tapLen</i> and <i>frameSize</i> are specified in the function ippsFullbandControllerInit_EC
<i>pSrcSin</i>	Pointer to the send-in signal frame. The frame size is specified in the function <code>ippsFullbandControllerInit_EC</code> .
<i>pDstStepSize</i>	Pointer to the vector of step sizes. Vector length is equal to <i>frameSize</i> specified in the function <code>ippsFullbandControllerInit_EC</code> .

Description

The functions `ippsFullbandControllerUpdate_EC` are declared in the `ippsc.h` file. These functions update the internal state of fullband controller and return the step size values for use in the coefficients' adaptation process.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <i>pState</i> , <i>pRin</i> , <i>pSin</i> , or <i>pStepSize</i> pointer is NULL.

FullbandController_EC

The main fullband controller function which updates filter coefficients and returns output gain coefficients.

Syntax

```
IppStatus ippsFullbandController_EC_32f(const Ipp32f* pAdaptiveFilterErr, const
    Ipp32f* pFixedFilterErr, Ipp32f* pAdaptiveCoefs, Ipp32f* pFixedCoefs,
    Ipp32f* pSGain, IppsFullbandControllerState_EC_32f* pState);

IppStatus ippsFullbandController_EC_16s(const Ipp16s* pAdaptiveFilterErr, const
    Ipp16s* pFixedFilterErr, Ipp16s* pAdaptiveCoefs, Ipp16s* pFixedCoefs,
    Ipp32s* pSGain, IppsFullbandControllerState_EC_16s* pState);
```

Parameters

<i>pState</i>	Pointer to the fullband controller state structure.
<i>pAdaptiveFilterErr</i>	Pointer to the vector of adaptive filter output. Vector length is equal to <i>frameSize</i> , specified in ippsFullbandControllerInit_EC .
<i>pFixedFilterErr</i>	Pointer to the vector of fixed filter output. Vector length is equal to <i>frameSize</i> , specified in ippsFullbandControllerInit_EC .
<i>pAdaptiveCoefs</i>	Pointer to the adaptive filter coefficients vectors. Vector length is equal to <i>tapLen</i> , specified in ippsFullbandControllerInit_EC .
<i>pFixedCoefs</i>	Pointer to the fixed filter coefficients vectors. Vector length is equal to <i>tapLen</i> , specified in ippsFullbandControllerInit_EC .
<i>pSGain</i>	Pointer to the send gain coefficient.

Description

The functions `ippsFullbandController_EC` are declared in the `ippsc.h` file.

These functions compare powers of adaptive filter error and fixed filter error and track changes of adaptive filter coefficients power.

If the adaptive filter has a significantly smaller error and is stable (which is indicated by coefficients power changing slowly), its coefficients are copied to the fixed filter. This case corresponds to “no double-talk” mode.

If the fixed filter has a significantly smaller error, its coefficients are copied to the adaptive filter. This case corresponds to the double-talk mode.

Send gain coefficient calculation (non-linear processor technology) is based on presence of the double-talk mode and changes of receive-in and send-in signal powers.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.

FullbandControllerReset_EC

Resets fullband controller state.

Syntax

```
IppStatus ippFullbandControllerReset_EC_32f(IppsFullbandControllerState_EC_32f*  
    pState);  
IppStatus ippFullbandControllerReset_EC_16s(IppsFullbandControllerState_EC_16s*  
    pState);
```

Parameters

<i>pState</i>	Pointer to the fullband controller state structure.
---------------	---

Description

The functions `ippFullbandControllerReset_EC` are declared in the `ippsc.h` file. These functions reset the fullband controller state. You can use these functions after an interruption in adaptation process occurred.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.

FIR_EC

Computes FIR filter results.

Syntax

```
IppStatus ippsFIR_EC_16s(const Ipp16s* pSrcSpchRef, const Ipp16s* pSrcSpch,
    Ipp16s* pDstSpch, int len, Ipp16s* pSrcTaps, Int tapsLen);

IppStatus ippsFIR_EC_32f(const Ipp32f* pSrcSpchRef, const Ipp32f* pSrcSpch,
    Ipp32f* pDstSpch, int len, Ipp32f* pSrcTaps, Int tapsLen);
```

Parameters

<i>pSrcSpchRef</i>	Pointer to the source original receive-out signal (R_{out}) of length ($tapsLen + len$).
<i>pSrcSpch</i>	Pointer to the send-in signal (S_{in}) with echo path.
<i>pDstSpch</i>	Pointer to the destination send-out signal (S_{out}) which is echo-free.
<i>len</i>	Length of source and destination signals.
<i>pSrcTaps</i>	Vector of FIR filter taps.
<i>tapsLen</i>	Number of taps in the FIR filter.

Description

The functions ippsFIR_EC are declared in the ippsc.h file.

These function perform finite impulse response (FIR) filtering.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.

NLMS_EC

Performs FIR filtering with coefficients update.

Syntax

```
IppStatus ippsNLMS_EC_16s(const Ipp16s* pSrcSpchRef, const Ipp16s* pSrcSpch,
    const Ipp32s* pStepSize, Ipp16s* pSrcDstErr, Ipp16s* pDstSpch, int len,
    Ipp16s* pSrcDstTaps, int tapsLen);

IppStatus ippsNLMS_EC_32f(const Ipp32f* pSrcSpchRef, const Ipp32f* pSrcSpch,
    const Ipp32f* pStepSize, Ipp32f* pSrcDstErr, Ipp32f* pDstSpch, int len,
    Ipp32f* pSrcDstTaps, int tapsLen);
```

Parameters

<i>pSrcSpchRef</i>	Pointer to the source original receive-out signal (R_{out}) of length ($tapsLen + len$).
<i>pSrcSpch</i>	Pointer to the send-in signal (S_{in}) with echo path.
<i>pStepSize</i>	Pointer to the vector of step sizes. Vector length is equal to <i>len</i> .
<i>pSrcDstErr</i>	Pointer to the last error value.
<i>pDstSpch</i>	Pointer to the destination send-out signal (S_{out}) which is echo-free.
<i>len</i>	Length of source and destination signals.
<i>pSrcDstTaps</i>	Source and destination vector of FIR filter taps.
<i>tapsLen</i>	Number of taps.

Description

The functions `ippsNLMS_EC` are declared in the `ippsc.h` file.

These function perform FIR filtering with normalized Least Mean Square (LMS) coefficients adaptation.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.

G722 Sub-Band ADPCM Speech Codec Related Functions

The Intel IPP functions described later in this section if properly combined may be used to construct the G.722 speech codec compliant to the reference implementation given by ITU-T library (Rec.192, Clause 7: *The ITU-T 64, 56 and 48 kbps wide-band speech coding algorithm*). The description of G.722 speech codec also known as Sub-Band ADPCM compression may be found in ITU-T G.722 Recommendation: *7KHz coding within 64 kbps*.

The Intel IPP functions described below are primarily concerned with the well-defined, computationally expensive core operations that comprise the G.722 subband codec.

The list of Intel IPP Subband ADPCM related functions is given in the following table:

Table 9-11 Intel IPP Subband ADPCM Related Functions

Function Base Name	Operation
<u>SBADPCMEncodeStateSize_G722</u>	Returns the memory size required for G.722 Sub-Band ADPCM encoder.
<u>SBADPCMEncodeInit_G722</u>	Initializes the memory buffer for the Sub-Band ADPCM encoder.
<u>SBADPCMEncode_G722</u>	Performs Sub-Band ADPCM encoding of the synthesis QMF samples.
<u>QMFEncode_G722</u>	Performs QMF analysis of the uniform PCM input.
<u>SBADPCMDecodeStateSize</u>	Returns the memory size required for Sub-Band ADPCM decoder.
<u>SBADPCMDecodeInit_G722</u>	Initializes the memory buffer for the Sub-Band ADPCM decoder.
<u>SBADPCMDecode_G722</u>	Performs decoding of the Sub-Band ADPCM compressed bit-stream.
<u>QMFDecode_G722</u>	Performs QMF synthesis of recovered samples.

SBADPCMEncodeStateSize_G722

Returns the memory size required for G.722 Sub-Band ADPCM encoder.

Syntax

```
IppStatus ippsSBADPCMEncodeStateSize_G722_16s (int* pEncMemSize)
```

Parameters

pEncMemSize Pointer to the output memory size in bytes.

Description

The function `ippsSBADPCMEncodeStateSize_G722` is declared in the `ippsc.h` file.

This function gets the information about the amount of memory (in bytes) needed to process the G.722 Sub-Band ADPCM compression.

Return Values

`ippStsNoErr` Indicates no error.

`IppStsNullPtrErr` Indicates an error when the *pEncMemSize* pointer is NULL.

SBADPCMEncodeInit_G722

Initializes the memory buffer for the Sub-Band ADPCM encoder.

Syntax

```
IppStatus ippsSBADPCMEncodeInit_G722_16s (IppsEncoderState_G722_16s* pEncMem)
```

Parameters

pEncMem Pointer to the input memory buffer of size required to properly initialize the G.722 Sub-Band ADPCM encoder.

Description

The function `ippsSBADPCMEncodeInit_G722_16s` is declared in the `ippsc.h` file.

This function initializes the memory buffer referenced by the pointer *pEncMem*. This buffer is required to enable G.722 Sub-Band ADPCM compression (encoding) starting from the reset state.

Return Values

`ippStsNoErr` Indicates no error.

`IppStsNullPtrErr` Indicates an error when the *pEncMem* pointer is NULL.

SBADPCMEncode_G722

Performs Sub-Band ADPCM encoding of the synthesis QMF samples.

Syntax

```
IppStatus ippsSBADPCMEncode_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst, int
    len, IppsEncoderState_G722_16s* pEncMem)
```

Parameters

<i>pSrc</i>	Pointer to the input vector of synthesis QMF samples.
<i>pDst</i>	Pointer to the ADPCM bit-stream output vector.
<i>len</i>	The length of input vector, must be a multiple of two.
<i>pEncMem</i>	Pointer to the state memory of the Sub-Band ADPCM encoder.

Description

The function `ippsSBADPCMEncode_G722_16s` is declared in the `ippsc.h` file.

This function performs the Sub-Band ADPCM compression (encoding) of the synthesized Quadrature Mirror Filter (QMF) samples within a bit rate of 64 kbps. The function uses the state memory referenced by *pEncMem* to encode the input data and update it accordingly. Before using this function, you must initialize the state memory by calling the [ippsSBADPCMEncodeInit_G722](#) function.

The function `ippsSBADPCMEncode_G722_16s` splits the frequency band of the input signal into two sub-bands (higher and lower) and encodes the signals in each sub-band using ADPCM compression. Each element of the output vector contains six binary digits of the lower sub-band and two binary digits of the higher sub-band for the 64 kbps bit rate ADPCM encoding.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not a multiple of two.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pMemUpdated</i> , <i>pSrc</i> , or <i>pDst</i> pointer is NULL.

QMFEncode_G722

Performs QMF analysis of the uniform PCM input.

Syntax

```
IppStatus ippsQMFEncode_G722_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,  
                                Ipp16s* delay)
```

Parameters

<i>pSrc</i>	Pointer to the uniform PCM input speech vector.
<i>pDst</i>	Pointer to the synthesis QMF samples output vector.
<i>len</i>	The length of input/output vectors, must be a multiple of two.
<i>delay</i>	Pointer to the buffer of a delay line.

Description

The function `ippsQMFEncode_G722_16s` is declared in the `ippsc.h` file.

This function uses Quadrature Mirror Filtering (QMF) of the 14-bit uniform PCM speech input to compute the lower and higher sub-band signal components for the Sub-Band ADPCM encoder. Computations are done in accordance with ITU-T Recommendation G.722.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not a multiple of two.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>delay</i> pointer is NULL.

SBADPCMDecodeStateSize

Returns the memory size required for Sub-Band ADPCM decoder.

Syntax

```
IppStatus ippsSBADPCMDecodeStateSize_G722 (int* pDecMemSize)
```

Parameters

<i>pDecMemSize</i>	Pointer to the output size of the memory needs to perform G.722 SB-ADPCM decoder.
--------------------	---

Description

The function `ippsSBADPCMDecodeStateSize_G722` is declared in the `ippsc.h` file.

This function gets the information about the amount of memory (in bytes) needed to process the G.722 Sub-Band ADPCM decompression.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDecMemSize</i> pointer is NULL.

SBADPCMDecodeInit_G722

Initializes the memory buffer for the Sub-Band ADPCM decoder.

Syntax

```
IppStatus ippsSBADPCMDecodeInit_G722_16s (IppsDecoderState_G722_16s* pDecMem)
```

Parameters

<i>pDecMem</i>	Pointer to the input memory buffer of size required to properly initialize the G.722 Sub-Band ADPCM decoder.
----------------	--

Description

The function `ippsSBADPCMDecodeInit_G722_16s` is declared in the `ippsc.h` file.

This function initializes the memory buffer referenced by the pointer `pDecMem`. This buffer is required to enable G.722 Sub-Band ADPCM decompression (decoding) starting from the reset state.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDecMem</code> pointer is NULL.

SBADPCMDecode_G722

Performs decoding of the Sub-Band ADPCM compressed bit-stream.

Syntax

```
IppStatus ippsSBADPCMDecode_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst, int
    len, Ipp16s mode, IppsDecoderState_G722_16s* pDecMem)
```

Parameters

<code>pSrc</code>	Pointer to the input vector that contains the even number of samples. The even element of this vector contains six binary digits of the lower sub-band and the odd element of this vector contains two binary digits of the higher sub-band.
<code>pDst</code>	Pointer to the output vector that contains decoded low-band and high-band portions of the recovered samples.
<code>len</code>	Length of the input vector.
<code>mode</code>	Decode bit rate of the G.722 decoder. The values of <code>mode</code> have the following meaning: <code>mode = 1</code> - sets 64 kbps bit rate <code>mode = 2</code> - sets 56 kbps bit rate <code>mode = 3</code> - sets 48 kbps bit rate

pDecMem Pointer to the state memory of the Sub-Band ADPCM decoder.

Description

The function `ippsSBADPCMDecode_G722_16s` is declared in the `ippsc.h` file.

This function decodes the lower-band and upper-band portions of the Sub-Band ADPCM compressed bit-stream into the low-band and upper-band portions of the recovered samples, respectively. This operation is the reverse of the Sub-Band ADPCM compression (see [ippsSBADPCMEncode_G722](#)). Note that the effective audio coding bit rate varies between 64, 56, or 48 kbps depending on the mode of operation.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pDecMem</i> , <i>pSrc</i> , or <i>pDst</i> pointer is NULL.

QMFDecode_G722

Performs QMF synthesis of recovered samples.

Syntax

```
IppStatus ippsQMFDecode_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst, int len,
                                   Ipp16s* delay);
```

Parameters

<i>pSrc</i>	Pointer to the input vector that contains decoded low-band and high-band portions of the recovered samples.
<i>pDst</i>	Pointer to the 16-bit linear PCM speech output vector.
<i>len</i>	The length of input/output vectors, must be a multiple of two.
<i>delay</i>	Pointer to the buffer of a delay line.

Description

The function `ippsQMFDecode_G722_16s` is declared in the `ippsc.h` file.

This function performs Quadrature Mirror Filter (QMF) synthesis of the low-band and high-band portions of the recovered samples to reconstruct the output signal. Computations are done in accordance with ITU-T Recommendation G.722.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>len</i> is than less or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not a multiple of two.
<code>IppStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> or <i>delay</i> pointer is NULL.

Companding Functions

The functions described in this section perform an operation of data compression by using a logarithmic encoder-decoder, referred to as companding. Companding allows you to maintain a constant percentage error by logarithmically spacing the quantization levels [[Rab78](#)].

The Intel IPP companding functions perform the following conversion operations of signal samples:

- From 8-bit μ -law encoded format to PCM-linear or vice-versa.
- From 8-bit A-law encoded format to PCM-linear or vice-versa.
- From 8-bit μ -law encoded format to A-law encoded or vice-versa.

Samples encoded in μ -law or A-law format are non-uniformly quantized. The quantization functions used by these formats are designed to reduce the dependency of signal-to-noise ratio on the magnitude of the encoded signal. This is achieved by quantization (companding) at a finer resolution near zero, and at a coarse resolution at larger positive or negative levels. The output values are normalized to be in the range $[-1; +1]$.

These functions perform the μ -law and A-law companding in compliance with the CCITT G.711 specification. For the conversion rules and more details, refer to [[CCITT](#)].

[Example 9-1](#) shows how to use companding functions.

MuLawToLin

Decodes samples from 8-bit μ -law encoded format to linear samples.

Syntax

```
IppStatus ippsMuLawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMuLawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector, which stores 8-bit μ -law encoded signal samples to be decoded.
<i>pDst</i>	Pointer to the destination vector, which stores the linear sample results.

len Number of samples in the vector.

Description

The function `ippsMuLawToLin` is declared in the `ipps.h` file. This function decodes the 8-bit μ -law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

LinToMuLaw

Encodes the linear samples using 8-bit μ -law format and stores them in a vector.

Syntax

```
ippStatus ippsLinToMuLaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
ippStatus ippsLinToMuLaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the vector that holds the signal samples (normalized to be less than 1.0) to be encoded.
<i>pDst</i>	Pointer to the vector that holds the output of the function <code>ippsLinToMuLaw</code> .
<i>len</i>	Number of samples in the vector.

Description

The function `ippsLinToMuLaw` is declared in the `ipps.h` file. This function encodes the PCM-linear samples in the vector *pSrc* using 8-bit μ -law format and stores them in the vector *pDst*.

The formula for μ -law companding is as follows:

$$|C_{\mu}(x)| = \frac{\ln(1 + 255 \cdot |x|)}{\ln(256)} \cdot 128, \quad -1 \leq x \leq 1$$

where x is the linear signal sample and $C_{\mu}(x)$ is the μ -law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

Application Notes

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of μ -law format is usually performed using look-up Tables 2a/G.711 and 2b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

[Example 9-1](#) shows how to use the function `ippsLinToMuLaw_32f8u`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pSrc</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

ALawToLin

Decodes the 8-bit A-law encoded samples to linear samples.

Syntax

```

IppStatus ippsALawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsALawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);

```

Parameters

`pSrc` Pointer to the vector that holds the signal samples to be converted.

<i>pDst</i>	Pointer to the vector that holds the output of the function <code>ippsALawToLin</code> .
<i>len</i>	Number of samples in the vector.

Description

The function `ippsALawToLin` is declared in the `ipps.h` file. This function decodes the 8-bit A-law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*.

[Example 9-1](#) shows how to use the function `ippsALawToLin_8u32f`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

LinToALaw

Encodes the linear samples using 8-bit A-law format and stores them in an array.

Syntax

```
IppStatus ippsLinToALaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);  
IppStatus ippsLinToALaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the vector that holds the signal samples to be encoded.
<i>pDst</i>	Pointer to the vector that holds the output of the function <code>ippsLinToALaw</code> .
<i>len</i>	Number of samples in the vector.

Description

The function `ippsLinToALaw` is declared in the `ipps.h` file. This function encodes the PCM-linear samples in the vector `pSrc` using 8-bit A-law format and stores them in the vector `pDst`.

The formula for A-law companding is as follows:

$$|C_A(x)| = \begin{cases} \frac{87.56|x|}{1 + \ln 87.56} \cdot 128, & 0 \leq |x| \leq \frac{1}{87.56} \\ \frac{1 + \ln(87.56|x|)}{1 + \ln 87.56} \cdot 128, & \frac{1}{87.56} < |x| \leq 1 \end{cases},$$

where x is the linear signal sample and $C_A(x)$ is the A-law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

Application Notes

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of A-law format is usually performed using look-up Tables 1a/G.711 and 1b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

MuLawToALaw

Converts samples from 8-bit μ -law encoded format to 8-bit A-law encoded format.

Syntax

```
IppStatus ippsMuLawToALaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector, which stores 8-bit μ -law encoded signal samples.
<i>pDst</i>	Pointer to the destination vector, which stores the 8-bit A-law encoded samples.
<i>len</i>	Number of samples in the vector.

Description

The function `ippsMuLawToALaw` is declared in the `ipps.h` file. This function converts signal samples from 8-bit μ -law encoded format in the vector *pSrc* to 8-bit A-law encoded format and stores them in the vector *pDst*.

Application Notes

The conversion of μ -law format to A-law format is usually performed using look-up Table 3/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

[Example 9-1](#) shows how to use the function `ippsMuLawToALaw_8u`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

ALawToMuLaw

Converts samples from 8-bit A-law encoded format to 8-bit μ -law encoded format.

Syntax

```
IppStatus ippsALawToMuLaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector, which stores 8-bit A-law encoded signal samples.
<i>pDst</i>	Pointer to the destination vector, which stores the 8-bit μ -law encoded samples.
<i>len</i>	Number of samples in the vector.

Description

The function `ippsMuLawToALaw` is declared in the `ipps.h` file. This function converts signal samples from 8-bit A-law encoded format in the vector *pSrc* to 8-bit μ -law format and stores them in the vector *pDst*.

Application Notes

The conversion of A-law format to μ -law format is usually performed using look-up Table 4/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is NULL.

`ippStsSizeErr` Indicates an error when *len* is less than or equal to 0.

Example 9-1 Using Companding Functions

```
void compand( void ) {  
    Ipp32f x[4] = { 0.1f, 0.2f, 0.3f, 0.4f };  
    Ipp8u m[4], a[4];  
    ippsLinToMuLaw_32f8u( x, m, 4 );  
    ippsMuLawToALaw_8u( m, a, 4 );  
    ippsALawToLin_8u32f( a, x, 4 );  
    // now x must be close to original  
    printf_32f("x =", x, 4, ippStsNoErr);  
}
```

Output:

```
x =  0.099609 0.207031 0.304688 0.398438
```

Audio Coding Functions

10

The subset of Intel® IPP for audio coding includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio encoder and decoder (see [\[ISO14496\]](#)), MP3 encoder and decoder. These functions implement pipeline blocks with large computational complexity.

The current set of functions is sufficient to implement a portable optimized MPEG-4 AAC Main profile decoder and a portable optimized MPEG-1, 2 Layer III encoder (see [\[ISO11172\]](#) and [\[ISO13818\]](#)).

Also this chapter includes Spectral Band Replication (SBR) functions as an audio coding bandwidth extension technology, which is standardized in ISO/IEC 14496-3:2001/Amd.1:2003 [\[ISO14496\]](#).

The full list of functions is given in [Table 10-1](#).

Table 10-1 Intel IPP Audio Coding Functions

Function Base Name	Operation
Interleaved to Multi-Row Format Conversion Functions	
Interleave	Converts signal from non-interleaved to interleaved format.
Deinterleave	Converts signal from interleaved to non-interleaved format.
Spectral Data Prequantization Functions	
Pow34	Raises a vector to the power of 3/4.
Pow43	Raises a vector to the power of 4/3.
Scale Factors Calculation Functions	
CalcSF	Restores actual scale factors from the bit stream values.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
Mantissa Conversion and Scaling Functions	
<u>Scale</u>	Applies scale factors to spectral bands in accordance with spectral bands boundaries.
<u>MakeFloat</u>	Converts mantissa and exponent arrays to float arrays.
Modified Discrete Cosine Transform Functions	
<u>MDCTFwdInitAlloc,</u> <u>MDCTInvInitAlloc</u>	Create and initialize modified discrete cosine transform specification structure.
<u>MDCTFwdInit, MDCTInvInit</u>	Initialize modified discrete cosine transform specification structure.
<u>MDCTFwdFree, MDCTInvFree</u>	Close modified discrete cosine transform specification structure.
<u>MDCTFwdGetSize, MDCTInvGetSize</u>	Get the size of MDCT specification structure, initialization, and work buffer.
<u>MDCTFwdGetBufSize,</u> <u>MDCTInvGetBufSize</u>	Get the size of MDCT work buffer.
<u>MDCTFwd, MDCTInv</u>	Compute forward or inverse MDCT of a signal.
Block Filtering Functions	
<u>FIRBlockInitAlloc</u>	Initializes FIR block filter state.
<u>FIRBlockFree</u>	Closes FIR block filter state.
<u>FIRBlockOne</u>	Filters vector of sample through FIR block filter.
Frequency Domain Prediction Functions	
<u>FDPInitAlloc</u>	Creates and initializes predictor state.
<u>FDPInit</u>	Intializes FDP state.
<u>FDPFree</u>	Closes FDP state.
<u>FDPGetSize</u>	Gets size of FDP state structure.
<u>FDPReset</u>	Resets predictors for all spectral lines.
<u>FDPResetSfb</u>	Resets predictor-specific information in some scale factor bands.
<u>FDPResetGroup</u>	Resets predictors for group of spectral lines.
<u>FDPFwd</u>	Performs frequency domain prediction procedure and calculates prediction error.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
FDPInv	Retrieves input signal from prediction error, using frequency domain prediction procedure.
VLC Functions	
VLCDecodeEscBlock_MP3	Parses the bitstream and decodes variable length code for MP3.
VLCDecodeEscBlock_AAC	Parses the bitstream and decodes variable length code for AAC.
VLCCountEscBits_MP3	Calculates the number of bits necessary for encoding in MP3 format.
VLCCountEscBits_AAC	Calculates the number of bits necessary for encoding in AAC format.
VLCEncodeEscBlock_MP3	Encodes an array of values into destination bitstream in MP3 format and advances bitstream pointer.
VLCEncodeEscBlock_AAC	Encodes an array of values into destination bitstream in AAC format and advances bitstream pointer.
Psychoacoustic Functions	
Spread	Computes spreading function.
Vector Quantization Functions	
VQCodeBookInitAlloc	Creates and initializes the codebook structure.
VQCodeBookInit	Initializes the codebook structure.
VQCodeBookFree	Closes the <code>IppsVQCodeBookState_32f</code> structure created by the function <code>VQCodeBookInitAlloc</code> .
VQCodeBookGetSize	Gets size of the codebook structure.
VQPreliminarySelect	Selects candidates for the nearest code vector of codebooks.
VQMainSelect	Finds optimal indexes with minimal distortion.
VQIndexSelect	Finds optimal vector set for specified number of codebooks.
VQReconstruction	Reconstructs vectors from indexes.
MP3 Encoder Functions	
AnalysisPQMF_MP3	Implements stage 1 of MP3 hybrid analysis filterbank.
MDCTFwd_MP3	Implements stage 2 of the MP3 hybrid analysis filterbank.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
<u>PsychoacousticModelTwo_MP3</u>	Implements the ISO/IEC 11172-3 psych-acoustic model recommendation 2 to estimate the masked threshold and perceptual entropy associated with a block of PCM audio input.
<u>JointStereoEncode_MP3</u>	Transforms independent left and right channel spectral coefficient vectors into combined mid/side (MS) and/or intensity (IS) mode coefficient vectors suitable for quantization.
<u>Quantize_MP3</u>	Quantizes the spectral coefficients generated by the analysis filterbank.
<u>PackScalefactors_MP3</u>	Applies noiseless coding to the scale factors and then packs the output into the bitstream buffer.
<u>HuffmanEncode_MP3</u>	Applies lossless Huffman encoding to the quantized samples and packs the output into the bitstream buffer.
<u>PackFrameHeader_MP3</u>	Packs the content of the frame header into the bitstream.
<u>PackSideInfo_MP3</u>	Packs the side information into the bitstream buffer.
<u>BitReservoirInit_MP3</u>	Initializes all elements of the bit reservoir state structure.
MP3 Decoder Functions	
<u>UnpackFrameHeader_MP3</u>	Unpacks the audio frame header.
<u>UnpackSideInfo_MP3</u>	Unpacks the side information from the input bitstream for use during the decoding of the associated frame.
<u>UnpackScaleFactors_MP3</u>	Unpacks scale factors.
<u>HuffmanDecode_MP3</u>	Decodes Huffman data.
<u>HuffmanDecodesfb_MP3</u>	
<u>HuffmanDecodesfbMbp_MP3</u>	
<u>ReQuantize_MP3</u>	Requantizes the decoded Huffman symbols.
<u>ReQuantizeSfb_MP3</u>	
<u>MDCTInv_MP3</u>	Performs the first stage of hybrid synthesis filter bank.
<u>SynthPQMF_MP3</u>	Performs the second stage of hybrid synthesis filter bank.
AAC Decoder Functions	
<u>UnpackADIFHeader_AAC</u>	Gets the AAC ADIF format header.
<u>UnpackADTSFrameHeader_AAC</u>	Gets ADTS frame header from the input bitstream.
<u>DecodePrgCfgElt_AAC</u>	Gets program configuration element from the input bitstream.
<u>DecodeChanPairElt_AAC</u>	Gets <i>channel_pair_element</i> from the input bitstream.
<u>NoiselessDecoder_LC_AAC</u>	Decodes all the data for one channel.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
<u>DecodeDatStrElt_AAC</u>	Gets data stream element from the input bitstream.
<u>DecodeFillElt_AAC</u>	Gets the fill element from the input bitstream.
<u>QuantInv_AAC</u>	Performs inverse quantization of Huffman symbols for current channel in-place.
<u>DecodeMsStereo_AAC</u>	Processes MS stereo for pair channels in-place.
<u>DecodeIsStereo_AAC</u>	Processes intensity stereo for pair channels.
<u>DeinterleaveSpectrum_AAC</u>	Deinterleaves the coefficients for short block.
<u>DecodeTNS_AAC</u>	Decodes for Temporal Noise Shaping in-place.
<u>MDCTInv_AAC</u>	Maps the time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples.
<u>DecodeMainHeader_AAC</u>	Gets main header information and main layer information from bit stream.
<u>DecodeExtensionHeader_AAC</u>	Get extension header information and extension layer information from bit stream.
<u>DecodePNS_AAC</u>	Implements perceptual noise substitution coding within an ICS.
<u>DecodeMsPNS_AAC</u>	Implements perceptual noise substitution coding in the case of joint coding.
<u>DecodeChanPairElt_MP4_AAC</u>	Gets <i>channel_pair_element</i> from the input bitstream
<u>LongTermReconstruct_AAC</u>	Uses Long Term Reconstruct to reduce the redundancy of a signal between successive coding frames.
<u>MDCTFwd_AAC</u>	Generates spectrum coefficient of PCM samples.
<u>EncodeTNS_AAC</u>	Performs reversion of TNS in the Long Term Reconstruct loop in-place.
<u>LongTermPredict_AAC</u>	Gets the predicted time domain signals in the Long Term Reconstruct (LTP) loop.
<u>NoiselessDecode_AAC</u>	Performs noiseless decoding.
<u>LtpUpdate_AAC</u>	Performs required buffer update in the Long Term Reconstruct (LTP) loop.
Spectral Band Replication Functions	
<u>AnalysisFilterGetSize_SBR</u>	Returns size of <code>FilterSpec_SBR</code> specification structures, init and work buffers.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
<u>SynthesisFilterGetSize_SBR</u>	Returns size of <code>FilterSpec_SBR</code> specification structures, init and work buffers.
<u>SynthesisDownFilterGetSize_SBR</u>	Returns size of <code>FilterSpec_SBR</code> specification structures, init and work buffers.
<u>AnalysisFilterInit_SBR</u>	Initializes analysis specification structure.
<u>SynthesisFilterInit_SBR</u>	Initializes synthesis specification structure.
<u>SynthesisDownFilterInit_SBR</u>	Initializes downsample synthesis specification structure.
<u>AnalysisFilter_SBR</u>	Splits time domain signal output from the core decoder into 32 subband signals.
<u>SynthesisFilter_SBR</u>	Transforms SBR-processed subband signals into time domain samples.
<u>SynthesisDownFilter_SBR</u>	Transforms SBR-processed subband signals into time domain samples and performs downsampling at the same time.
<u>PredictCoef_SBR</u>	Obtains prediction filter coefficients using covariance method.

Interleaved to Multi-Row Format Conversion Functions

This section describes functions that enable you to perform transformations between interleaved and non-interleaved forms of multichannel signal. A signal in the interleaved form is a vector with interleaved samples for different channels. In the non-interleaved form, samples for each channel are stored in a separate vector.



NOTE. Although you may use both aligned and unaligned memory for arrays, you should expect slower performance when memory is not aligned.

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Interleave

Converts signal from non-interleaved to interleaved format.

Syntax

```
IppStatus ippsInterleave_16s(const Ipp16s** pSrc, int ch_num, int len,
                             Ipp16s* pDst);
IppStatus ippsInterleave_32f(const Ipp32f** pSrc, int ch_num, int len,
                             Ipp32f* pDst);
```

Parameters

<i>pSrc</i>	Array of pointers to the vectors [<i>len</i>] containing samples for particular channels.
<i>ch_num</i>	Number of channels.
<i>len</i>	Number of samples in each channel.
<i>pDst</i>	Pointer to the destination vector [<i>ch_num</i> * <i>len</i>] in interleaved format.

Description

This function is declared in the `ippac.h` header file. The function `ippsInterleave` transforms the signal from the non-interleaved to interleaved format according to the formula

$$pDst[i] = pSrc[i \text{ div } ch_num][i \text{ mod } ch_num], 0 \leq i < ch_num * len,$$

where `div` is the integer part of the quotient and `mod` is the remainder.

The function `ippsInterleave_32f` is used in the float-point version of MP3 encoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsNumChannelsErr</code>	Indicates an error when <code>ch_num</code> is less than or equal to 0.
<code>ippStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.

Deinterleave

Converts signal from interleaved to non-interleaved format.

Syntax

```

IppStatus ippsDeinterleave_16s(const Ipp16s* pSrc, int ch_num, int len,
                               Ipp16s** pDst);

IppStatus ippsDeinterleave_32f(const Ipp32f* pSrc, int ch_num, int len,
                               Ipp32f** pDst);

```

Parameters

<code>pSrc</code>	Pointer to vector [<code>ch_num * len</code>] of interleaved samples.
<code>ch_num</code>	Number of channels.
<code>len</code>	Number of samples in each channel.

pDst Array of pointers to the vectors [*len*] to be filled with samples of particular channels.

Description

This function is declared in the `ippac.h` header file. The function `ippsDeinterleave` transforms the input signal from interleaved to non-interleaved format according to the formula

$$pDst[i][j] = pSrc[i + j * ch_num], 0 \leq i < ch_num, 0 \leq j < len.$$

The function `ippsInterleave_16s` is used in the AAC and MP3 encoders and the function `ippsInterleave_32f` is used in the AC3 decoder (float-point version), and AAC and MP3 encoders (float-point version) included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> or <i>ch_num</i> is less than or equal to 0.
<code>ippStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.

Spectral Data Prequantization Functions

MPEG-1, 2 Layer III and MPEG-4 AAC audio encoders raise the spectral data to the power of 3/4 before quantization to provide a more consistent signal-to-noise ratio over the range of quantized values. The re-quantizers in the decoders linearize the values by raising their output to the power of 4/3.

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm> .

Pow34

Raises a vector to the power of 3/4.

Syntax

```
IppStatus ippsPow34_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int len);
IppStatus ippsPow34_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPow34_16s_Sfs(const Ipp16s* pSrc, int inscaleFactor, int
    Ipp16s* pDst, int scaleFactor, int len);
```

Parameters

<i>pSrc</i>	Pointer to the input data vector [<i>len</i>].
<i>inscaleFactor</i>	Input data scalefactor value.
<i>pDst</i>	Pointer to the output data vector [<i>len</i>].
<i>scaleFactor</i>	Scalefactor value.
<i>len</i>	Number of elements in the input and output vectors.

Description

This function is declared in the `ippac.h` header file. The function `ippsPow34` performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = |pSrc[i]|^{3/4}, 0 \leq i < len$$

and stores the result in *pDst*.

For example, in MPEG-1 Layer III the quantization of the complete vector of spectral values is done according to the following formula:

$$ix(i) = nint\left(\left(\frac{|xr(i)|}{\sqrt[4]{2^{qquant + quantanf}}}\right)^{0.75} - 0.0946\right), \text{ where}$$

ix is the array of quantized values, *i* is the number of values in the array, *nint* is the function used to round non-integer values to the nearest integer value, *xr* is the vector of the magnitudes of the spectral values, *qquant* is the quantizer step size information, and *quantanf* is a constant that depends on the spectral flatness measure.

You can use the function `ippsPow34_32f16s` for this operation. However, if the maximum of all quantized values is outside the table range or the overall bit sum exceeds the available bits, you should repeat this operation several times with a new *qquant* value and the same *xr* array.

Alternatively, you can use the function `ippsPow34_32f` to calculate the power of 3/4 for *xr* only once before the inner iteration loop and use it during every quantization iteration with the constant multiplier and convert the resulting values to `short`.

$$multiplier = \left(\frac{1}{\sqrt[4]{2^{qquant + quantanf}}}\right)^{0.75}.$$

This operation is supported by the function [ippsMulC_Low_32f16s](#).

The function `ippsPow34` is used in the float-point versions of AAC and MP3 encoders included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippsStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Pow43

Raises a vector to the power of 4/3.

Syntax

```
IppStatus ippsPow43_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the input data vector [<i>len</i>].
<i>pDst</i>	Pointer to the output data vector [<i>len</i>].
<i>len</i>	Number of elements in the input and output vectors.

Description

This function is declared in the `ippac.h` header file. The function `ippsPow43` performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = \text{sign}(pSrc[i]) * |pSrc[i]|^{\frac{4}{3}}, 0 \leq i < len$$

and stores the result in *pDst*.

The function `ippsPow43` is used in the float-point versions of AAC and MP3 decoders included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.



NOTE. The input values in the *pSrc* array should not be greater than 8206. The function does not check if the input values meet this requirement in order not to cause performance slowdown. If the input values in the array exceed the limit, the function may operate incorrectly. No error code is returned in this case.

Scale Factors Calculation Functions

In MPEG-2, 4 GA AAC decoder scale factors extracted from the bitstream require an additional restoring procedure. The function `CalcSF` restores actual scale factors from values transmitted in the bit stream.

CalcSF

*Restores actual scale factors
from the bit stream values.*

Syntax

```
IppStatus ippsCalcSF_16s32f(const Ipp16s* pSrc, int offset, Ipp32s* pDst,  
    int len);
```

Parameters

<i>pSrc</i>	Pointer to the input data array.
<i>offset</i>	Scale factors offset.
<i>pDst</i>	Pointer to the output data array.
<i>len</i>	Number of elements in the vector.

Description

This function is declared in the `ippac.h` header file. The function `ippsCalcSF` restores actual scale factors from the values `pSrc` transmitted in the bit stream, using the common scale factor offset. Computation is performed according to the following formula

$$pDst[i] = 2^{\frac{1}{4}(pSrc[i]-offset)}, 0 \leq i < len.$$

Restored scale factors are written into `pDst`.

The function `ippsCalcSF` is used in the float-point versions of AAC and MP3 decoders included into IPP Samples downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.



NOTE. The input values in the `pSrc` array should be in the range $-512 \leq pSrc[i] - offset < 512$. Otherwise the function may operate incorrectly. No error message is returned in this case.

Mantissa Conversion and Scaling Functions

The scale application procedure is necessary for MPEG-2, 4 GA AAC decoder to restore the original spectral values from a set of the inversely quantized spectral coefficients. The whole spectrum is divided into a set of scale factor bands. Since the widths differ from band to band, the band positions are defined by the offset vector.

The mantissa conversion procedure is necessary for the AC3 decoder to restore the original spectral values from a set of the exponents and mantissas in the bitstream.

Scale

Applies scale factors to spectral bands in accordance with spectral bands boundaries.

Syntax

```
IppStatus ippsScale_32f_I(Ipp32f* pSrcDst, const Ipp32f* pSF,  
    const int* pBandOffset, int bandsNumber);
```

Parameters

<i>pSrcDst</i>	Pointer to the input and output data array. The size of array must be not less than <i>pBandOffset[bands_number]</i> .
<i>pSF</i>	Pointer to the data array containing scale factors. The size of the array must be not less than <i>bands_number</i> .
<i>pBandOffset</i>	Pointer to the vector of band offsets. The size of array must be not less than <i>bands_number</i> + 1.
<i>bandsNumber</i>	Number of bands to which scale factors are applied.

Description

This function is declared in the `ippac.h` header file. The function `ippsScale_32f_I` computes scaled values from the input vector, the vector of scale factors, and the vector of band offsets. Operations are performed in-place.

This function applies the set of scale factors *pSF* with *bandsNumber* elements to the bands constructed from the input vector *pSrc*. Band boundaries are defined by the vector of the band offsets *pBandOffset*. All values in each band are multiplied by the corresponding scale factor.



NOTE. The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of *pSrcDst* vector is contained in the *pBandsOffset[bandsNumber]* element.

The function `ippsScale` is used in the float-point versions of AAC and MP3 decoders included into IPP Samples downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm> .

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> or <i>pBandsOffset</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>bandsNumber</i> is less than or equal to 0.

MakeFloat

Converts mantissa and exponent arrays to float arrays.

Syntax

```
IppStatus ippMakeFloat_16s32f (Ipp32s* inmant, Ipp32s* inexp, Ipp32s
    size, Ipp32f* outfloat);
```

Parameters

<i>inmant</i>	Array of mantissas.
<i>inexp</i>	Array of exponents.
<i>size</i>	Number of array elements.
<i>outfloat</i>	Array of resulting float arrays.

Description

This function is declared in the `ippac.h` header file. This function converts the mantissa and exponent arrays decoded from the bitstream to the float array of spectral samples by the formula:

$$outfloat[i] = inmant[i] \times 2^{-inexp[i] - 15},$$

where $i = 0 \dots size$.

The conversion serves to improve application performance when decoding bitstreams in the AC3 format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>inmant</i> , <i>inexp</i> or <i>outfloat</i> pointer is NULL.

`ippStsSizeErr` Indicates an error when *size* is less than or equal to 0.

Modified Discrete Cosine Transform Functions

This section describes Intel IPP functions that compute the modified discrete cosine transform (MDCT) of a signal. MDCT is a lapped orthogonal transform widely used in different audio codecs, such as MPEG-1, MPEG-2, AC-3, and AAC.

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

MDCTFwdInitAlloc, MDCTInvInitAlloc

Create and initialize modified discrete cosine transform specification structure.

Syntax

```
IppStatus ippMDCTFwdInitAlloc_32f(IppsMDCTFwdSpec_32f** ppMDCTSpec,
    int len);

IppStatus ippMDCTFwdInitAlloc_16s(IppsMDCTFwdSpec_32s** ppMDCTSpec,
    int len);

IppStatus ippMDCTInvInitAlloc_32f(IppsMDCTInvSpec_32f** ppMDCTSpec,
    int len);
```

Parameters

<i>ppMDCTSpec</i>	Pointer to pointer to MDCT specification structure to be created.
<i>len</i>	Number of samples in MDCT. Since this set of functions was designed specially for audio coding, only the following values of length are supported: 12, 36, and 2^k , where $k \geq 5$. These values are the only values that appear in audio coding.

Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdInitAlloc` and `ippsMDCTInvInitAlloc` create and initialize MDCT specification structure *pMDCTSpec* with the specified transform length *len*.

The function `ippsMDCTFwdInitAlloc` initializes the forward MDCT specification structure. The function `ippsMDCTInvInitAlloc` initializes the inverse MDCT specification structure.

`ippsMDCTFwdInitAlloc_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders,

`ippsMDCTFwdInitAlloc_16s` is used in the fixed-point version of AAC encoder,

`ippsMDCTInvInitAlloc_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppMDCTSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> does not belong to the above set of admissible values.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.



NOTE. [ippsMDCTFwd_16s_Sfs](#) does not support the length values of 12 and 36.

MDCTFwdInit, MDCTInvInit

Initialize modified discrete cosine transform specification structure.

Syntax

```
IppStatus ippsMDCTFwdInit_32f(IppsMDCTFwdSpec_32f** ppMDCTSpec, int len,
    Ipp8u* pMemSpec, Ipp8u* pMemInit);
```

```

IppStatus ippsMDCTFwdInit_16s(IppsMDCTFwdSpec_32f** ppMDCTSpec, int len,
    Ipp8u* pMemSpec, Ipp8u* pMemInit);
IppStatus ippsMDCTInvInit_32f(IppsMDCTInvSpec_32f** ppMDCTSpec, int len,
    Ipp8u* pMemSpec, Ipp8u* pMemInit);

```

Parameters

<i>ppMDCTSpec</i>	Pointer to pointer to MDCT specification structure to be created.
<i>len</i>	Number of samples in MDCT. Since this set of functions was designed specially for audio coding, only the following values of length are supported: 12, 36, and 2^k , where $k \geq 5$. These values are the only values that appear in audio coding.
<i>pMemSpec</i>	Pointer to the area for MDCT specification structure.
<i>pMemInit</i>	Pointer to the buffer that is used for initialization.

Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdInit` and `ippsMDCTInvInit` initialize MDCT specification structure *ppMDCTSpec* with the specified transform length *len*.

The function `ippsMDCTFwdInit` initializes the forward MDCT specification structure. The function `ippsMDCTInvInit` initializes the inverse MDCT specification structure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppMDCTSpec</i> , <i>pMemSpec</i> or <i>pMemInit</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> does not belong to the above set of admissible values.



NOTE. [`ippsMDCTFwd_16s_sfs`](#) does not support the length values of 12 and 36.

MDCTFwdFree, MDCTInvFree

Closes modified discrete cosine transform specification structure.

Syntax

```
IppStatus ippMDCTFwdFree_32f(IppsMDCTFwdSpec_32f* pMDCTSpec);
IppStatus ippMDCTFwdFree_16s(IppsMDCTFwdSpec_16s* pMDCTSpec);
IppStatus ippMDCTInvFree_32f(IppsMDCTInvSpec_32f* pMDCTSpec);
```

Parameters

pMDCTSpec Pointer to the MDCT specification structure to be closed.

Description

These functions are declared in the `ippac.h` header file. The functions `ippMDCTFwdFree` and `ippMDCTInvFree` close the MDCT structure *pMDCTSpec* by freeing all memory associated with the specification created by `ippMDCTFwdInitAlloc` or `ippMDCTInvInitAlloc` functions.

Call either `ippMDCTFwdFree` or `ippMDCTInvFree` after the transform is completed.

The function `ippMDCTFwdFree` closes the forward MDCT specification structure.

The function `ippMDCTInvFree` closes the inverse MDCT specification structure.

`ippMDCTFwdFree_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders,

`ippMDCTFwdFree_16s` is used in the fixed-point version of AAC encoder,

`ippMDCTInvFree_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMDCTSpec</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pMDCTSpec</i> is incorrect.

MDCTFwdGetSize, MDCTInvGetSize

Get the sizes of MDCT specification structure, MDCT initialization, and MDCT work buffer.

Syntax

```
IppStatus ippsMDCTFwdGetSize_32f(int len, int* pSizeSpec, int*
    pSizeInit, int* pSizeBuf);
IppStatus ippsMDCTFwdGetSize_16s(int len, int* pSizeSpec, int*
    pSizeInit, int* pSizeBuf);
IppStatus ippsMDCTInvGetSize_32f(int len, int* pSizeSpec, int*
    pSizeInit, int* pSizeBuf);
```

Parameters

<i>len</i>	Number of samples in MDCT. Since this set of functions is designed specially for audio coding, only the following values of length are supported: 12, 36, and 2^k , where $k \geq 5$. These values are the only values that appear in audio coding.
<i>pSizeSpec</i>	Address of the MDCT specification structure size value in bytes.
<i>pSizeInit</i>	Address of the MDCT initialization buffer size value in bytes.
<i>pSizeBuf</i>	Address of the MDCT work buffer size value in bytes.

Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdGetSize` and `ippsMDCTInvGetSize` get the specification structure, initialization, and work buffer sizes of the MDCT described by the length and store the result in *pSizeSpec*, *pSizeInit*, and *pSizeBuf*.

The function `ippsMDCTFwdGetSize` gets the size of the work buffer for the forward MDCT. The function `ippsMDCTInvGetSize` gets the size of the work buffer for the inverse MDCT.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSizeSpec</i> , <i>pSizeInit</i> , or <i>pSizeBuf</i> pointer is NULL.

`ippStsSizeErr`

Indicates an error when *len* does not belong to the above set of admissible values.



NOTE. [`ippMDCTFwd_16s_sfs`](#) does not support the length values of 12 and 36.

MDCTFwdGetBufSize, MDCTInvGetBufSize

Gets the size of MDCT work buffer.

Syntax

```
IppStatus ippMDCTFwdGetBufSize_32f(const IppsMDCTFwdSpec_32f*
    pMDCTSpec, int* pSize);
IppStatus ippMDCTFwdGetBufSize_16s(const IppsMDCTFwdSpec_16s*
    pMDCTSpec, int* pSize);
IppStatus ippMDCTInvGetBufSize_32f(const IppsMDCTInvSpec_32f*
    pMDCTSpec, int* pSize);
```

Parameters

<i>pMDCTSpec</i>	Pointer to the MDCT specification structure.
<i>pSize</i>	Address of the MDCT work buffer size value in bytes.

Description

These functions are declared in the `ippac.h` header file. The functions `ippMDCTFwdGetBufSize` and `ippMDCTInvGetBufSize` get the work buffer size of the MDCT described by the specification structure *pMDCTSpec* and store the result in *pSize*.

The function `ippMDCTFwdGetBufSize` gets the size of the work buffer for the forward MDCT.

The function `ippMDCTInvGetBufSize` gets the size of the work buffer for the inverse MDCT.

`ippsMDCTGetBufSize_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders,
`ippsMDCTFwdGetBufSize_16s` is used in the fixed-point version of AAC encoder,
`ippsMDCTInvGetBufSize_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMDCTSpec</code> pointer or <code>pSize</code> value is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification structure <code>pMDCTSpec</code> is invalid.

MDCTFwd, MDCTInv

Computes forward or inverse modified discrete cosine transform (MDCT) of a signal.

Syntax

```

IppStatus ippsMDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsMDCTFwdSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsMDCTInvSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTFwd_32f_I(Ipp32f* pSrcDst, const IppsMDCTFwdSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f_I(Ipp32f* pSrcDst, const IppsMDCTInvSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTFwd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsMDCTFwdSpec_32f* pMDCTSpec, int scaleFactor, Ipp8u* pBuffer);

```

Parameters

<code>pSrc</code>	Pointer to the input data array.
<code>pDst</code>	Pointer to the output data array.
<code>pSrcDst</code>	Pointer to the input and output data array for the in-place operations.

<i>pMDCTSpec</i>	Pointer to the MDCT specification structure.
<i>pBuffer</i>	Pointer to the MDCT work buffer.
<i>scaleFactor</i>	Scalefactor value.

Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwd` and `ippsMDCTInv` compute the forward and inverse modified discrete cosine transform (MDCT), respectively.

In the following definition of MDCT, N denotes the length and $n_0 = (N/2 + 1)/2$.

For the forward MDCT, $x(n)$ is `pSrc[n]` and $y(k)$ is `pDst[k]`, whereas for the inverse MDCT $x(n)$ is `pDst[n]` and $y(k)$ is `pSrc[k]`.

The forward MDCT is defined by the formula:

$$y(k) = 2 \cdot \sum_{n=0}^{N-1} x(n) \cos\left(\frac{\pi}{N}(n + n_0)(2k + 1)\right), \text{ for } 0 \leq k < \frac{N}{2}.$$

The inverse MDCT is defined as

$$x(n) = \frac{2}{N} \cdot \sum_{k=0}^{\frac{N}{2}-1} y(k) \cos\left(\frac{\pi}{N}(n + n_0)(2k + 1)\right), \text{ for } 0 \leq n < N.$$

The *pBuffer* argument provides the MDCT functions with the necessary work memory and helps to avoid memory allocation within the functions. The buffer may also increase the performance if the MDCT functions use the result of the previous operation stored in cache as an input array.

`ippsMDCTFwd_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders, `ippsMDCTInv_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders, `ippsMDCTFwd_16s_sfs` is used in the fixed version of AAC encoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

<code>ippStsContextMatchErr</code>	Indicates an error when the specification structure <i>pMDCTSpec</i> is invalid.
<code>ippStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.

Block Filtering Functions

Intel IPP functions described in this section implement the finite impulse response (FIR) block filter. You can use this group of functions to design transform domain adaptive filters. These filters preprocess the signal by decomposing the input vector into orthogonal components, which are subsequently used as inputs to a parallel bank of adaptive subfilters. You may use this approach for implementing frequency domain linear predictors in audio codecs, for example, CELP and AAC.

The filtering function receives a number of vectors (signals). Every call of a filtering function produces one filtered sample for each input signal. The library functions do not perform any particular adaptation method but you can specify the filter taps at each call of a filtering function.

To use the FIR block filter functions, follow these general steps:

1. Call [FIRBlockInitAlloc](#) to initialize the state structure of a block filter.
2. Call [FIRBlockOne](#) to filter a vector of samples through a block filter.
3. Call [FIRBlockFree](#) to free dynamic memory associated with the FIR block filter.

FIRBlockInitAlloc

Initializes FIR block filter state.

Syntax

```
ippStatus ippFIRBlockInitAlloc_32f(IppsFIRBlockState_32f** pState,
    int order, int len);
```

Parameters

<i>pState</i>	Pointer to the FIR block filter state structure to be created.
<i>order</i>	Number of elements in the array containing the tap values.
<i>len</i>	Number of input signals.

Description

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockInitAlloc` creates and initializes a FIR block filter state.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are NULL.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>order</i> or <i>len</i> is less than or equal to 0.

FIRBlockFree

Closes FIR block filter state.

Syntax

```
IppStatus ippsFIRBlockFree_32f(IppsFIRBlockState_32f* pState);
```

Parameters

pState Pointer to the FIR block filter state structure to be closed.

Description

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockFree` closes the FIR block filter state by freeing all memory associated with the filter state created by the function `ippsFIRBlockInitAlloc`.

Call `ippsFIRBlockFree` after filtering is completed.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pState</i> is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

FIRBlockOne

*Filters vector of samples
through FIR block filter.*

Syntax

```
IppStatus ippsFIRBlockOne_32f(Ipp32f* pSrc, Ipp32f* pDst,
    IppsFIRBlockState_32f* pState, Ipp32f* pTaps);
```

Parameters

<i>pSrc</i>	Pointer to the input vector of samples to be filtered.
<i>pDst</i>	Pointer to the vector of filtered output samples.
<i>pState</i>	Pointer to the FIR filter state structure.
<i>pTaps</i>	Pointer to the vector of filter taps.

Description

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockOne` filters a vector of samples *pSrc* of the length *len* through a filter and stores the result in *pDst*.

The filter taps are specified in the vector *pTaps* of the length *order*. The values of *len* and *order* parameters are specified in the [FIRBlockInitAlloc](#) call.

In the following definition of the FIR filter, the sample of the input vector *i* to be filtered with the delay *k* is denoted x_{n-k}^i , and the taps are denoted h_k . The output value y_i is defined by the following formula:

$$y_n^i = \sum_{k=0}^{order-1} h_k x_{n-k}^i, \quad 0 \leq i < len.$$

Before calling the function `ippsFIRBlockOne`, initialize the filter state by calling the function `ippsFIRBlockInitAlloc`. Specify the taps values in the argument *pTaps*.

Example 10-1 Single-Rate Filtering with the `ippsFIRBlockOne` Function

```
IppStatus fir(void)
{
    #undef NUMITERS
```

Example 10-1 Single-Rate Filtering with the `ippsFIRBlockOne` Function (continued)

```
#define NUMITERS 20
#undef BLOCKSIZE
#define BLOCKSIZE 20
    int n;
    int i;
    IppStatus status;
    IppsFIRBlockState_32f *fctx;
    Ipp32f x [BLOCKSIZE], y [BLOCKSIZE];
    const float taps[] = {
        0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f,
0.1869f,
        0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0
    };
    ippsFIRBlockInitAlloc_32f( &fctx, 11, BLOCKSIZE );
    for (n = 0; n < NUMITERS; ++n)
    {
        for (i = 0; i < BLOCKSIZE; i++) x[i] = (Ipp32f)sin(IPP_2PI *
            n * 0.2 + i);
        status = ippsFIRBlockOne_32f( x, y, fctx, (Ipp32f*)taps );
        for (i = 0; i < BLOCKSIZE; i++)
            printf("%f", y[i]);
    }
    ippsFIRBlockFree_32f(fctx);
    return status;
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.
<code>ippStsMisalignedBuf</code>	Indicates misaligned arrays. Supply aligned data for better performance.
<code>ippStsFIRLenErr</code>	Indicates an error when one of the following conditions is true: <i>pState->len</i> is less than or equal to 0 <i>pState->order</i> is less than or equal to 0 <i>pState->queue_end</i> is less than 0 <i>pState->queue_end</i> is greater or equal to <i>pState->order</i> .

Frequency Domain Prediction Functions

MPEG-2, 4 AAC encoder uses prediction in frequency domain (FDP) to decrease redundancy in the audio signal and ensure more effective coding. For each spectral line, input signals are filtered through the second order adaptive FIR filter called a predictor. Then, instead of processing the signal, the difference between the original signal and the filter output (that is, the prediction error), is passed for further processing. The decoder derives the original signal from the prediction error using a symmetrical block.

You should regularly reset the predictors to their initial state to reduce accumulated calculation error. You may also need to reset the predictors in special cases discussed and described in the ISO-144963 standard. You can reset predictors for the entire spectrum, several scale factor bands, or a selected group of spectral lines.

For more information on the algorithm of filter coefficient adaptation and FDP usage see ISO-144963, clause 6.5.3.2.

To use the FDP prediction tool functions described in this section, follow these steps:

1. Call the function [FDPInitAlloc](#) to allocate memory and initialize predictor state.
2. Call the function [FDPFwd](#) for each frame to calculate prediction error or call the function [FDPInv](#) to retrieve the original signal.
3. Call the functions [FDPReset](#), [FDPResetSfb](#), or [FDPResetGroup](#) to reset predictors in certain spectral lines at any time after creating the state.
4. Call the function [FDPFree](#) to free the memory allocated by `ippsFDPInitAlloc`.

FDPInitAlloc

Creates and initializes predictor state.

Syntax

```
IppStatus ippsFDPInitAlloc_32f(IppsFDPState_32f** ppFDPState, int len);
```

Parameters

<i>ppFDPState</i>	Pointer to pointer to the FDP state structure to be created.
<i>len</i>	Number of spectral lines to be processed.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPInitAlloc_32f` creates and initializes the FDP state.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>ppFDPState</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

FDPInit

Initializes predictor state.

Syntax

```
IppStatus ippsFDPInit_32f(IppsFDPState_32f** ppFDPState, int len, Ipp8u*
    pMemSpec);
```

Parameters

<code>ppFDPState</code>	Pointer to pointer to the FDP state structure to be initialized.
<code>len</code>	Number of spectral lines to be processed.
<code>pMemSpec</code>	Pointer to the area for FDP state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPInit_32f` initializes the FDP state.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>ppFDPState</code> or <code>pMemSpec</code> is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

`ippStsMemAllocErr` Indicates an error when no memory is allocated.

FDPFree

Closes FDP state.

Syntax

```
IppStatus ippSFDPFree_32f(IppsFDPState_32f* pFDPState);
```

Parameters

pFDPState Pointer to the FDP state structure to be closed.

Description

This function is declared in the `ippac.h` header file. The function `ippSFDPFree` closes the FDP state by freeing all memory associated with the FDP state structure created by the function [FIRBlockInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFDPState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

FDPGetSize

Gets size of FDP state structure.

Syntax

```
IppStatus ippSFDPGetSize_32f(int len, int* pSizeState);
```

Parameters

len Number of spectral lines to be processed.

pSizeState Address of size value in bytes of the frequency domain predictor state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPGetSize_32f` gets sizes of the frequency domain predictor state structure and stores the results in *pSizeState*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSizeState</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

FDPReset

Resets predictors for all spectral lines.

Syntax

```
ippStatus ippsFDPReset_32f(IppsFDPState_32f* pFDPState);
```

Parameters

pFDPState Pointer to the predictor specific state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPReset` resets predictors for all spectral lines.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFDPState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

FDPResetSfb

*Resets predictor-specific information
in some scale factor bands.*

Syntax

```
IppStatus ippSFDPResetSfb_32f (const int* pSfbOffset, int sfbNumber, const Ipp8u*  
    pResetFlag, IppsFDPState_32f* pFDPState);
```

Parameters

<i>pSfbOffset</i>	Pointer to the band offset vector.
<i>sfbNumber</i>	Number of bands.
<i>pResetFlag</i>	Array of flags showing whether predictors for spectral lines in a certain scale factor band need to be reset.
<i>pFDPState</i>	Pointer to the predictor specific state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippSFDPResetSfb_32f` resets predictors for all spectral lines in each scale factor band *i*, for which `pResetFlag[i]` is not equal to 0.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>sfbNumber</i> is less than or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

FDPResetGroup

Resets predictors for group of spectral lines.

Syntax

```
IppStatus ippSFDPResetGroup_32f (int resetGroupName, int step, IpsFDPState_32f*
    pFDPState);
```

Parameters

<i>resetGroupName</i>	Number of the group to be reset.
<i>step</i>	Distance between two neighboring spectral lines in the group.
<i>pFDPState</i>	Pointer to the predictor specific state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippSFDPResetGroup_32f` resets predictors for each *step*-th spectral line beginning from the start up to the end of spectrum.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFDPState</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>resetGroupName</i> or <i>step</i> is less than or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

FDPFwd

Performs frequency domain prediction procedure and calculates prediction error.

Syntax

```
IppStatus ippSFDPFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, IpsFDPState_32f*
    pFDPState);
```

Parameters

<i>pSrc</i>	Pointer to the input data array.
<i>pDst</i>	Pointer to the data array to be filled with prediction errors.
<i>pFDPState</i>	Pointer to the predictor specific state structure.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsFDPFwd` applies frequency domain prediction procedure to the input signal *pSrc* and stores prediction errors in the *pDst* vector.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFDPState</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.
<code>ippStsMisalignedBuf</code>	Indicates misaligned arrays. Supply aligned data for better performance.

FDPIInv

*Retrieves input signal from prediction error,
using frequency domain prediction procedure.*

Syntax

```
IppStatus ippsFDPInv_32f_I(Ipp32f* pSrcDst, const int* pBandsOffset, int  
    predictorBandsNumber, Ipp8u* pPredictionUsed, IppsFDPState_32f* pFDPState);
```

Parameters

<i>pSrcDst</i>	Pointer to the input and output data array for the in-place operation.
<i>pBandsOffset</i>	Pointer to the band offset vector.
<i>predictorBandsNumber</i>	Number of scale factor bands.

<i>pPredictionUsed</i>	Pointer to array of flags showing whether prediction will be used in certain scale factor band.
<i>pFDPState</i>	Pointer to the predictor specific state structure.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsFDPInv` applies the procedure of frequency domain prediction to specific bands of the input spectral vector *pSrcDst*. Positions of bands are defined by the parameters *predictorBandsNumber* and *pBandsOffset*.

For each scale factor band *i*, if *pPredictionUsed[i]* is not equal to 0, all values of *pSrcDst* within the band are treated as a prediction error and the original signal is restored. If *pPredictionUsed[i]* is equal to 0, all values of *pSrcDst* within the band are treated as a signal and will be passed without any changes.

Regardless of the *pPredictionUsed* flag, the coefficients of each predictor are updated.



NOTE. The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of *pSrcDst* vector is stored in the *pBandsOffset[predictorBandsNumber]* element.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>predictorBandsNumber</i> is less than or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.
<code>ippStsMisalignedBuf</code>	Indicates misaligned arrays. Supply aligned data for better performance.

VLC Functions

Variable Length Coding (VLC) is an audio data compression method that uses statistical modeling to define which values occur more frequently than others to build tables for subsequent encoding and decoding operations. Audio data in the bitstream is encoded with Variable Length Code tables so that the shortest codes correspond to the most frequent values and the longer codes correspond to the less frequent values. Every standard that uses Variable Length Coding has corresponding tables named `inputTable` that list possible codes and their values.

Example 10-2 inputTable Format

```
static IppsVLCtable_32s inputTable[] =
{
    {value0, code0, length0};
    {value1, code1, length1};
    ...
    {valueN, codeN, lengthN};
}
```

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

VLCDecodeEscBlock_MP3

Parses the bitstream and decodes variable length code for MP3.

Syntax

```
IppStatus ippsVLCDecodeEscBlock_MP3_lul6s(const Ipp8u** ppBitStream, int*
    pBitOffset, int linbits, Ipp16s* pData, int len, const
    IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

<i>ppBitStream</i>	Pointer to pointer to the current byte in the bitstream buffer. <i>*ppBitStream</i> is updated by the function.
<i>pBitOffset</i>	Pointer to the bit position in the byte pointed by <i>*ppBitStream</i> . <i>*pBitOffset</i> is updated by the function.
<i>linbits</i>	Length of escape sequence.

<i>pData</i>	Pointer to the array of decoded values.
<i>len</i>	Number of values to decode into array <i>pData</i> .
<i>pVLCSpec</i>	Pointer to VLCDecoder specification structure.

Description

The function `ippsVLCDecodeEscBlock_MP3_1u16s` is declared in the `ippac.h` header file. The function parses the bitstream and decodes variable length code using the VLCDecoder specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function `ippsVLCDecodeEscBlock_MP3_1u16s`, rearrange *inputTable* before initialization as follows:

```
value = (first_sample_value) << 8) + (second_sample_value + 128).
code = (((code_ini << (first_sample_value != 0)) + (first_sample_value <
0)) << (second_sample_value != 0)) + (first_sample_value < 0).
length = length_ini + (first_sample_value != 0) + (second_sample_value !=
0).
```

MP3Esc VLC codebook is an unsigned two-dimensional codebook with a large absolute value equal to 15.

From one record of the initial *inputTable* you can obtain from one to four records of rearranged *inputTable*.

Example

```
abs(first_sample_value) = 9;
abs(second_sample_value) = 6;

code_ini = 0xf5
length_ini = 12;

record 1: first_sample_value = -9, second_sample_value = -6;
    value = (-9 << 8) + (-6 + 128) = -2182;
    code = (((0xf5 << 1) + 1) << 1) + 1 = 0x3d7;
    length = 12 + 1 + 1 = 14;
    {-2182, 0x3d7, 14)

record 2: first_sample_value = -9, second_sample_value = 6;
    value = (-9 << 8) + (6 + 128) = -2170;
    code = (((0xf5 << 1) + 1) << 1) + 0 = 0x3d6;
```

```
length = 12 + 1 + 1 = 14;  
-2170, 0x3d6, 14)
```

```
record 3: first_sample_value = 9, second_sample_value = -6;  
value = (9 << 8) + (-6 + 128) = 2426;  
code = (((0xf5 << 1) + 0) << 1) + 1 = 0x3d5;  
length = 12 + 1 + 1 = 14;  
{2426, 0x3d7, 14)
```

```
record 4: first_sample_value = 9, second_sample_value = 6;  
value = (9 << 8) + (6 + 128) = 2438;  
code = (((0xf5 << 1) + 0) << 1) + 0 = 0x3d4;  
length = 12 + 1 + 1 = 14;  
{2438, 0x3d7, 14)
```

The function `ippsVLCDecodeEscBlock_MP3_1u16s` is used in the float-point and fixed-point versions of MP3 decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers passed to the function is NULL.
<code>ippStsBitOffsetErr</code>	Indicates an error when <i>*pBitOffset</i> is less than 0 or more than 7.
<code>ippStsContextMatchErr</code>	Indicates an error when <code>pVLCSpec</code> structure was not created by <code>ippsVLCDecodeInit_32s</code> or <code>ippsVLCDecodeInitAlloc_32s</code> functions.
<code>ippStsVLCInputDataErr</code>	Indicates an error when incorrect input is used. For decode functions it can indicate that bitstream contains code that is not specified inside the used table.

VLCDecodeEscBlock_AAC

Parses the bitstream and decodes variable length code for AAC.

Syntax

```
IppStatus ippsVLCDecodeEscBlock_AAC_1u16s(const Ipp8u** ppBitStream, int*
    pBitOffset, Ipp16s* pData, int len, const IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

<i>ppBitStream</i>	Pointer to pointer to the current byte in the bitstream buffer. <i>*ppBitStream</i> is updated by the function.
<i>pBitOffset</i>	Pointer to the bit position in the byte pointed by <i>*ppBitStream</i> . <i>*pBitOffset</i> is updated by the function.
<i>pData</i>	Pointer to the array of decoded values.
<i>len</i>	Number of values to decode into array <i>pData</i> .
<i>pVLCSpec</i>	Pointer to VLCDecoder specification structure.

Description

The function `ippsVLCDecodeEscBlock_AAC_1u16s` is declared in the `ippac.h` header file. The function parses the bitstream and decodes variable length code using the VLCDecoder specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function `ippsVLCDecodeEscBlock_AAC_1u16s`, rearrange *inputTable* before initialization as follows:

```
value = (first_sample_value << 8) + (second_sample_value + 128).
code = (((code_ini << (first_sample_value != 0)) + (first_sample_value <
0)) << (second_sample_value != 0)) + (first_sample_value < 0).
length = length_ini + (first_sample_value != 0) + (second_sample_value !=
0).
```

AACEsc VLC codebook is an unsigned two-dimensional codebook with a large absolute value is equal to 16.

From one record of initial *inputTable* you can obtain from one to four records of rearranged *inputTable*.

Example

```
Initial input table record - {159, 0x1ad, 9}
abs(first_sample_value) = 159/17 = 9;
abs(second_sample_value) = 159 - 17*9 = 6;

code_ini = 0x1ad
length_ini = 9;

record 1: first_sample_value = -9, second_sample_value = -6;
        value = (-9 << 8) + (-6 + 128) = -2182;
        code = (((0x1ad << 1) + 1) << 1) + 1 = 0x6b7;
        length = 9 + 1 + 1 = 11;
        {-2182, 0x6b7, 11)

record 2: first_sample_value = -9, second_sample_value = 6;
        value = (-9 << 8) + (6 + 128) = -2170;
        code = (((0x1ad << 1) + 1) << 1) + 0 = 0x6b6;
        length = 9 + 1 + 1 = 11;
        {-2170, 0x6b6, 11)

record 3: first_sample_value = 9, second_sample_value = -6;
        value = (9 << 8) + (-6 + 128) = 2426;
        code = (((0x1ad << 1) + 0) << 1) + 1 = 0x6b5;
        length = 9 + 1 + 1 = 11;
        {2426, 0x6b7, 11)

record 4: first_sample_value = 9, second_sample_value = 6;
        value = (9 << 8) + (6 + 128) = 2438;
        code = (((0x1ad << 1) + 0) << 1) + 0 = 0x6b4;
        length = 9 + 1 + 1 = 11;
        {2438, 0x6b7, 11)
```

The function `ippSVLCDecodeEscBlock_AAC_1u16s` is used in the float-point and fixed-point versions of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers passed to the function is <code>NULL</code> .

<code>ippStsBitOffsetErr</code>	Indicates an error when <i>*pBitOffset</i> is less than 0 or more than 7.
<code>ippStsContextMatchErr</code>	Indicates an error when <code>pVLCSpec</code> structure was not created by <code>ippSVLCDecodeInit_32s</code> or <code>ippSVLCDecodeInitAlloc_32s</code> functions.
<code>ippStsVLCInputDataErr</code>	Indicates an error when incorrect input is used. For decode functions it can indicate that bitstream contains code that is not specified inside the used table.
<code>ippStsVLCAACEscCodeLengthErr</code>	Indicates an error when bitstream contains AAC-Esc code with the length more than 21.

VLCCountEscBits_MP3

Calculates the number of bits necessary for encoding in MP3 format.

Syntax

```
IppStatus ippSVLCCountEscBits_MP3_16s32s(const Ipp16s* pInputData, int len, int
linbits, Ipp32s* pCountBits, const IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

<i>pInputData</i>	Pointer to the array of source values.
<i>len</i>	Size of values array <i>pInputData</i> .
<i>linbits</i>	Length of escape sequence.
<i>pCountBits</i>	Pointer to calculated length in bits to encode <i>pInputData</i> .
<i>pVLCSpec</i>	Pointer to <code>VLCDecoder</code> specification structure.

Description

The function `ippSVLCCountEscBits_MP3_16s32s` is declared in the `ippac.h` header file. The function calculates the number of bits necessary for encoding source data in *pInputData*, using the `VLCDecoder` specification structure built by the function `ippSVLCDecodeInit_32s` or `ippSVLCDecodeInitAlloc_32s` from data compression domain. To use the function

`ippsVLCCountEscBits_MP3_16s32s`, rearrange *inputTable* before initialization in the same way as described in the description of [VLCDecodeEscBlock_MP3](#) with one exception (because the large absolute value is equal to 15):

`value= (first_sample_value << 5) + (second_sample_value + 15).`

The function `ippsVLCCountEscBits_MP3_16s32s` is used in the float-point and fixed-point versions of MP3 encoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when <code>pVLCSpec</code> structure was not created by <code>ippsVLC EncodeInit_32s</code> or <code>ippsVLC EncodeInitAlloc_32s</code> functions.

VLCCountEscBits_AAC

Calculates the number of bits necessary for encoding in AAC format.

Syntax

`IppStatus ippsVLCCountEscBits_AAC_16s32s(const Ipp16s* pInputData, int len, Ipp32s* pCountBits, const IppsVLC EncodeSpec_32s* pVLCSpec);`

Parameters

<i>pInputData</i>	Pointer to the array of source values.
<i>len</i>	Size of values array <i>pInputData</i> .
<i>pCountBits</i>	Pointer to calculated length in bits to encode <i>pInputData</i> .
<i>pVLCSpec</i>	Pointer to VLC Encoder specification structure.

Description

The function `ippsVLCCountEscBits_AAC_16s32s` is declared in the `ippac.h` header file. The function calculates the number of bits necessary for encoding source data in `pInputData`, using the `VLCDecoder` specification structure built by the function `ippsVLCEncodeInit_32s` or `ippsVLCEncodeInitAlloc_32s` from data compression domain. To use the function `ippsVLCCountEscBits_AAC_16s32s`, rearrange `inputTable` before initialization in the same way as described in the description of [VLCDecodeEscBlock_AAC](#) with one exception (because the large absolute value is equal to 16):

```
value = (first_sample_value << 6) + (second_sample_value + 16).
```

The function `ippsVLCCountEscBits_AAC_16s32s` is used in the float-point and fixed-point versions of AAC encoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when <code>pVLCSpec</code> structure was not created by <code>ippsVLCEncodeInit_32s</code> or <code>ippsVLCEncodeInitAlloc_32s</code> functions.

VLC Encode EscBlock_MP3

Encodes an array of values into destination bitstream in MP3 format and advances bitstream pointer.

Syntax

```
IppStatus ippsVLCEncodeEscBlock_MP3_16s1u(const Ipp16s* pInputData, int len,
int linbits, Ipp8u** ppBitStream, int* pBitOffset, const
IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

<code>pInputData</code>	Pointer to the array of source values.
<code>len</code>	Size of values array <code>pInputData</code> .
<code>linbits</code>	Length of escape sequence.

<i>ppBitStream</i>	Pointer to pointer to the current byte in the bitstream buffer. <i>*ppBitStream</i> is updated by the function.
<i>pBitOffset</i>	Ppointer to the bit position in the byte pointed by <i>*ppBitStream</i> . <i>*pBitOffset</i> is updated by the function.
<i>pVLCSpec</i>	Pointer to VLCDecoder specification structure.

Description

The function `ippsVLCDecodeEscBlock_MP3_16s1u` is declared in the `ippac.h` header file. The function decodes an array of values from *pInputData* into destination *Bitstream* and advances the bitstream pointer, using the VLCDecoder specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain. To use the function `ippsVLCCountEscBits_MP3_16s32s`, rearrange *inputTable* before initialization in the same way as described in the description of [VLCDecodeEscBlock_MP3](#) with one exception (because the large absolute value is equal to 15):
$$\text{value} = (\text{first_sample_value} \ll 5) + (\text{second_sample_value} + 15).$$

The function `ippsVLCDecodeEscBlock_MP3_16s1u` is used in the float-point and fixed-point versions of MP3 encoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are NULL.
<code>ippStsBitOffsetErr</code>	Indicates an error when <i>*pBitOffset</i> is less than 0 or more than 7.
<code>ippStsContextMatchErr</code>	Indicates an error when <i>pVLCSpec</i> structure was not created by <code>ippsVLCDecodeInit_32s</code> or <code>ippsVLCDecodeInitAlloc_32s</code> functions.

VLCEncodeEscBlock_AAC

Encodes an array of values into destination bitstream in AAC format and advances bitstream pointer.

Syntax

```
IppStatus ippsVLCEncodeEscBlock_AAC_16slu(const Ipp16s* pInputData, int len,
      Ipp8u** ppBitStream, int* pBitOffset, const IppsVLCDecodeSpec_32s*
      pVLCSpec);
```

Parameters

<i>pInputData</i>	Pointer to the array of source values.
<i>len</i>	Size of values array <i>pInputData</i> .
<i>ppBitStream</i>	Pointer to pointer to the current byte in the bitstream buffer. * <i>ppBitStream</i> is updated by the function.
<i>pBitOffset</i>	Pointer to the bit position in the byte pointed by * <i>ppBitStream</i> . * <i>pBitOffset</i> is updated by the function.
<i>pVLCSpec</i>	Pointer to VLCDecoder specification structure.

Description

The function `ippsVLCEncodeEscBlock_AAC_16slu` is declared in the `ippac.h` header file. The function encodes an array of values from *pInputData* into destination *Bitstream* and advance bitstream pointer, using the VLCDecoder specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain. To use the function `ippsVLCCountEscBits_AAC_16s32s`, rearrange *inputTable* before initialization in the same way as described in the description of [VLCDecodeEscBlock_AAC](#) with one exception (because the large absolute value is equal to 16):

$$\text{value} = (\text{first_sample_value} \ll 6) + (\text{second_sample_value} + 16).$$

The function `ippsVLCEncodeEscBlock_AAC_16slu` is used in the float-point and fixed-point versions of AAC encoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are NULL.
<code>ippStsBitOffsetErr</code>	Indicates an error when <i>*pBitOffset</i> is less than 0 or more than 7.
<code>ippStsContextMatchErr</code>	Indicates an error when <code>pVLCSpec</code> structure was not created by <code>ippsVLCDecodeInit_32s</code> or <code>ippsVLCDecodeInitAlloc_32s</code> functions.

Psychoacoustic Functions

This section describes psychoacoustic functions.

Spread

Computes spreading function.

Syntax

```
IppStatus ippsSpread_16s_Sfs(Ipp32s* Src1, Ipp32s Src2, int  
    inScaleFactor, Ipp32s* pDst);
```

Parameters

<i>Src1</i>	Input data 1.
<i>Src2</i>	Input data 2.
<i>inScaleFactor</i>	Input value scalefactor value.
<i>pDst</i>	Pointer to the output data vector, output data is in Q15 format.

Description

This function is declared in the `ippac.h` file. The function gives a masking threshold produced by a single tone masker frequency *Src2* (in Bark) for neighboring frequency *Src1* (in Bark). This function is widely used in psychoacoustics.

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when `pDst` pointer is NULL.

The function `ippsSpread_16s_sfs` is used in the fixed-point version of AAC encoder included into IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Vector Quantization Functions

This sections describes functions used for vector quantization operations.

VQCodeBookInitAlloc

Creates and initializes the codebook structure.

Syntax

```
ippStatus ippsVQCodeBookInitAlloc_32f(const Ipp32f* pInputTable,
    IppsVQCodeBookState_32f** ppCodeBook, int step, int height);
```

Parameters

<i>pInputTable</i>	Pointer to the codebook table of the size <i>step</i> * <i>height</i> containing <i>height</i> quantization vectors of the length <i>step</i> .
<i>ppCodeBook</i>	Pointer to pointer to the codebook structure to be created.
<i>step</i>	Step to the next line in the table <i>InputTable</i> , quantization vector length.
<i>height</i>	Table height, number of quantization vectors.

Description

This function is declared in the `ippac.h` header file. This function creates and initializes the structure that contains the codebook and additional information needed for the search operation. This structure is used during the vector quantization operation by the [VQPreliminarySelect](#), [VQMainSelect](#), [VQIndexSelect](#), and [VQReconstruction](#) functions.

To free the memory allocated by this function, use the function [VQCodeBookFree](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>ppCodeBook</code> or <code>pInputTable</code> pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsSizeErr</code>	Indicates an error when <code>step</code> or <code>height</code> is less than or equal to 0.

VQCodeBookInit

Initializes the codebook structure.

Syntax

```
IppStatus ippvVQCodeBookInit_32f(const Ipp32f* pInputTable,  
    IppsVQCodeBookState_32f** ppCodeBook, int step, int height, Ipp8u*  
    pMemSpec);
```

Parameters

<code>pInputTable</code>	Pointer to the codebook table of the size <code>step * height</code> containing <code>height</code> quantization vectors of the length <code>step</code> .
<code>ppCodeBook</code>	Pointer to pointer to the codebook structure to be created.
<code>step</code>	Step to the next line in the table <code>InputTable</code> , quantization vector length.
<code>height</code>	Table height, number of quantization vectors.
<code>pMemSpec</code>	Pointer to the area for the codebook structure.

Description

This function is declared in the `ippac.h` header file. This function initializes the structure that contains the codebook and additional information needed for the search operation. This structure is used during the vector quantization operation by the [VQPreliminarySelect](#), [VQMainSelect](#), [VQIndexSelect](#), and [VQReconstruction](#) functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pInputTable</i> , <i>ppCodeBook</i> or <i>pMemSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.

VQCodeBookFree

Closes the `IppsVQCodeBookState_32f` structure created by the function `ippsVQCodeBookInitAlloc`.

Syntax

```
IppStatus ippsVQCodeBookFree_32f( IppsVQCodeBookState_32f* pCodeBook );
```

Parameters

pCodeBook Pointer to specified `IppsVQCodeBookState_32f` structure.

Description

This function is declared in the `ippac.h` header file. The function closes the `IppsVQCodeBookState_32f` structure created by [VQCodeBookInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCodeBook</i> pointer is NULL.

VQCodeBookGetSize

Gets the size of the codebook structure.

Syntax

```
IppStatus ippsVQCodeBookGetSize_32f( int step, int height, int*
    pSizeState );
```

Parameters

<i>step</i>	Step to the next line in the codebook table, quantization vector length.
<i>height</i>	Table height, number of quantization vectors.
<i>pSizeState</i>	Address of the codebook structure size value in bytes.

Description

This function is declared in the `ippac.h` header file. This function gets the codebook structure sizes and stores the results in *pSizeState*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSizeState</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when the step or height is less than or equal to 0.

VQPreliminarySelect

Selects candidates for the nearest code vector of codebooks.

Syntax

```
IppStatus ippVQPreliminarySelect_32f(const Ipp32f* pSrc, const Ipp32f*  
    pWeights, int nDiv, const Ipp32s* pLengths, Ipp32s* pIndx, Ipps32s* pSign,  
    int nCand, int* pPolbits, IppsVQCodeBookState_32f* pCodeBook);
```

Parameters

<i>pSrc</i>	Source vector to be quantized.
<i>pWeights</i>	Pointer to the vector of weights.
<i>nDiv</i>	Number of fragmentations of the <i>src</i> and <i>weights</i> vectors.
<i>pLengths</i>	Pointer to an array of lengths of fragmentations.
<i>pIndx</i>	Pointer to the output vector of indexes of the <i>nCand</i> minimum candidates.
<i>pSign</i>	Pointer to the output vector of signs of <i>nCand</i> minimum candidates.

- The value of 1 indicates that the minimal distortion appears when the norm is negative.
- The value of 0 indicates that the minimal distortion appears when the norm is positive.

<i>nCand</i>	Number of output candidates.
<i>pPolbits</i>	Pointer to the <i>polbits</i> flag vector.
<i>pCodeBook</i>	Pointer to the specified <code>IppsVQCodeBookState_32f</code> structure.

Description

This function is declared in the `ippac.h` header file. The function computes indexes and finds *nCand* vectors with the closest values from the codebook.

$$dist_p[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] - pSrcDiv[ismp])^2$$

if the *polbits[idiv]* is greater than `MAXBIT_SHAPE`, the following distortion are also calculated

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

for *idiv* = 0 to *nDiv* - 1, *icb* is the number of the codebook line.

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

$$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCodeBook* initialized in the function [VQCodeBookInitAlloc](#).

The function then selects *nCand* candidates of *dist* and *dist_n* with the minimum distortion measure.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCodeBook</i> or <i>pSrc</i> pointer is NULL.

VQMainSelect

Finds optimal indexes with minimal distortion.

Syntax

```
IppStatus ippsvQMainSelect_32f(const Ipp32f* pSrc, const Ipp32f* pWeights, int
    nDiv, const Ipp32s* pLengths, int nCand, Ipp32s** ppIndexCand, Ipps32s**
    ppSignCand, Ipp32s** ppIndx, Ipp32s** ppSign, IppsVQCodeBookState_32f**
    ppCodeBooks, int nCodeBooks);
```

Parameters

<i>pSrc</i>	Source vector to be quantized.
<i>pWeights</i>	Pointer to the vector of weights.
<i>nDiv</i>	Number of fragmentations of the <i>src</i> and <i>weights</i> vectors.
<i>pLengths</i>	Pointer to an array of lengths of fragmentations.
<i>nCand</i>	Number of input candidates.
<i>ppIndexCand</i>	Pointer to pointer to the input vectors of indexes of <i>nCand</i> minimum candidates.
<i>ppSignCand</i>	Pointer to pointer to the input vectors of signs of <i>nCand</i> minimum candidates.
<i>ppIndx</i>	Pointer to pointer to the output vectors of indexes.
<i>ppSign</i>	Pointer to pointer to the output vectors of signs.

<i>ppCodeBooks</i>	Pointer to pointer to the specified <code>IppsVQCodeBookState_32f</code> structures.
<i>nCodeBooks</i>	Number of codebooks.

Description

This function is declared in the `ippac.h` header file. The function restores vectors for all possible combinations of indexes across the specified number of codebooks then computes the distortion against the source vector. The function then returns the combination that provides for the minimal distortion.

The following formula serves for computing of the distortion for the given fragmentation *idiv* with the specified quantization vectors *icb[i]*, where *i* is within the range of $[0, nCodeBooks-1]$.

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp](rec[ismp] - pSrcDiv[ismp])^2$$

where

$$rec[ismp] = \frac{\sum_{i=0}^{nCodeBooks-1} pSignCand[idiv*nCand + icb[i]] * ppTable[pIndexCand[idiv*nCand + icb[i]]][ismp]}{nCodeBooks}$$

In these formulas

- pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

$$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCodeBook* initialized in the function [VQCodeBookInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

VQIndexSelect

Finds optimal vector set for specified number of codebooks.

Syntax

```
IppStatus ippVQIndexSelect_32f(const Ipp32f* pSrc, const Ipp32f* pWeights, int
    nDiv, const Ipp32s* pLengths, int nCand, int** ppPolbits, Ipp32s** ppIndx,
    Ipp32s** ppSign, IppsVQCodeBookState_32f** ppCodeBooks, int nCodeBooks);
```

Parameters

<i>pSrc</i>	Source vector to be quantized.
<i>pWeights</i>	Pointer to the vector of weights.
<i>nDiv</i>	Number of fragmentations of the <i>src</i> and <i>weights</i> vectors.
<i>pLengths</i>	Pointer to an array of lengths of fragmentations.
<i>nCand</i>	Number of input candidates.
<i>ppPolbits</i>	Indicates whether one or two norms should be used to compute the optimal vector set for the specified number of codebooks.
<i>ppIndx</i>	Pointer to pointer to the output vectors of indexes.
<i>ppSign</i>	Pointer to pointer to the output vectors of signs. <ul style="list-style-type: none"> • The value of 1 indicates that the minimal distortion appears when the norm is negative. • The value of 0 indicates that the minimal distortion appears when the norm is positive.

<i>ppCodeBooks</i>	Pointer to pointer to the specified <code>IppsVQCodeBookState_32f</code> structures.
<i>nCodeBooks</i>	Number of codebooks.

Description

This function is declared in the `ippac.h` header file. The function computes *nCand* vectors for each codebook by the following formula

$$dist_p[idiv][icb] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] - pSrcDiv[ismp])^2$$

if *polbits[idiv]* is 1, the following distortion is also calculated

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

for *idiv*=0 to *nDiv* -1, *icb* – number of codebook line.

The function subsequently restores vectors for all possible combinations of indexes across the specified number of codebooks then computes the distortion against the source vector. The function then returns the combination that provides for the minimal distortion. The following formula serves for computing of the distortion for the given fragmentation *idiv* with the specified quantization vectors *icb[i]*, where *i* is within the range of $[0, nCodeBooks-1]$.

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp] (rec[ismp] - pSrcDiv[ismp])^2$$

$$rec[ismp] = \frac{\sum_{i=0}^{nCodeBooks-1} pSignCand[idiv * nCand + icb[i]] * ppTable[pIndexCand[idiv * nCand + icb[i]]][ismp]}{nCodeBooks}$$

for *idiv*=0 to *nDiv* -1, *icb* – number of codebook line.

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

$$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCodeBook* initialized in the function [VQCodeBookInitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.



NOTE. You can use this function instead of calling the functions [VQPreliminarySelect](#) and [VQMainSelect](#) to save time and memory cost.

VQReconstruction

Reconstructs vectors from indexes.

Syntax

```
IppStatus ippVQReconstruction_32f(const Ipp32s** ppIndx, const Ipp32s**
    ppSign, const Ipp32s* pLengths, int nDiv, Ipp32f* pDst,
    IppsVQCodeBookState_32f** ppCodeBooks, int nCodeBooks);
```

Parameters

<i>ppIndx</i>	Pointer to pointer to an array of input vectors of indexes for each codebook.
<i>ppSign</i>	Pointer to pointer to an array of input vectors of signs for each codebook containing either 1 or -1.
<i>pLengths</i>	Pointer to an array of lengths of partitions of output vector.
<i>pDst</i>	Pointer to the reconstructed vector of spectrum values.
<i>nDiv</i>	Number of partitions of output vector.
<i>ppCodeBooks</i>	Pointer to pointer to an array of pointers to specified <code>IppsVQCodeBookState_32f</code> structures.
<i>nCodeBooks</i>	Number of codebooks.

Description

This function is declared in the `ippac.h` header file. The function reconstructs a vector divided on several partitions. A partition is defined as set of indexes of vectors from the specified number of codebooks, that is, one index for one partition from each codebook. To reconstruct the values of the output vector, the function calculates the arithmetic mean of vector values from the specified number of codebooks. The arrays *pIndx* and *pSign* contain the number of vectors from the codebook and the sign.

$$pDst \left[\sum_{i=0}^{ndiv-1} pLength[i] + j \right] = \frac{1}{nCodeBooks} \sum_{k=0}^{nCodeBooks-1} pSign[k][i] \cdot pCodeBooks[k] \rightarrow table[pIndx[k][i]][j]$$

for $i=0$ to $nDiv - 1$ and for $j = 0$ to $pLength[i]-1$.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.



NOTE. Length of pDst array should be equal to

$$\sum_{k=0}^{nCodeBooks-1} pLengths[k]$$

MP3 Audio Coding Functions

The [ISO/IEC 11172-3](#) MPEG-1, Layer III (also referred to as “MP3”) audio coding algorithms are widely used to compress stereophonic and dual-channel music signals. Well-suited for both transmission and storage applications, the MP3 algorithm delivers high-fidelity audio playback quality with bitrates as low as one-tenth of the original. As a result, the MP3 algorithm has become the *de facto* standard compression methodology for portable and handheld storage media, as well as for transmission of high-fidelity compressed audio over the Internet. MP3 encoder/decoder is widely used in music storage and audio recording.

Macros and Constants

The MP3 macro and constant definitions are listed in [Table 10-2](#).

Table 10-2 **MP3 Macro and Constant Definitions**

Global Macro Name	Definition	Notes
IPP_MP3_GROANULE_LEN	576	The number of samples in one granule
IPP_MP3_V_BUF_LEN	512	V data buffers length (32-bit words)
IPP_MP3_SF_BUF_LEN	40	Scale factor buffer length (8-bit words)
IPP_MP3_SFB_TABLE_LONG_LEN	138	Scale factor band table for long block length (16-bit words)
IPP_MP3_SFB_TABLE_SHORT_LEN	84	Scale factor band table for short block length (16-bit words)
IPP_MP3_ID_MPEG2	0	MPEG-2 frame identifier
IPP_MP3_ID_MPEG1	1	MPEG-1 frame identifier

Data Structures

The MP3 coding API includes several data structures.

The structure `IppMP3FrameHeader` contains the complete set of header information associated with one frame.

The structure `IppMP3SideInfo` contains the complete set of side information associated with one granule of one channel.

The structure `IppMP3HuffmanTable` contains the complete set of information associated with Huffman table.

The structure `IppMP3PsychoacousticModelTwoAnalysis` contains the outputs generated by the Intel IPP implementation of [ISO/IEC 11172-3](#) psychoacoustic analysis model 2, including estimates of the masked thresholds and perceptual entropy associated with the current frame. Masked thresholds are represented in terms of Mask-to-Signal Ratios (MSRs).

The structure `IppMP3PsychoacousticModelTwoState` contains the state information associated with the Intel IPP implementation of [ISO/IEC 11172-3](#) psychoacoustic analysis model 2 to facilitate coherent block processing.

The structure `IppMP3BitReservoir` contains the state information associated with the quantization bit reservoir.

Frame Header

```
typedef struct {
    int id;                /* ID: 1 - MPEG-1, 0 - MPEG-2 */
    int layer;             /* layer index: 0x3 - Layer I
                           0x2 - Layer II
                           0x1 - Layer III */
    int protectionBit;     /* CRC flag: 0 - CRC on, 1 - CRC off */
    int bitRate;           /* bit rate index */
    int samplingFreq;      /* sampling frequency index */
    int paddingBit;        /* padding flag: 0 - no padding, 1 -
                           padding */
    int privateBit;        /* private_bit, not used */
    int mode;              /* mono/stereo selection */
    int modeExt;           /* extension to mode */
}
```

```
int copyright;          /* copyright or not: 0 - no, 1 - yes */
int originalCopy;       /* original or copied: 0 - copy,
                        1 - original */
int emphasis;          /* flag indicating the type of de-emphasis */
int CRCWord;           /* CRC-check word */

} IppMP3FrameHeader;
```

Side Information

```
typedef struct {
    int part23Len;       /* number of main_data bits */
    int bigVals;         /* half the number of Huffman code words whose
                        maximum amplitudes may be greater than 1 */
    int globGain;        /* quantizes step size information */
    int sfCompress;      /* number of bits used for scale factors */
    int winSwitch;       /* window switch flag */
    int blockType;       /* block type flag */
    int mixedBlock;      /* flag: 0 - non-mixed block, 1 - mixed
                        block */
    int pTableSelect[3]; /* Huffman table index for the 3 rectangle in
                        <big_values> field */
    int pSubBlkGain[3];  /* gain offset from the global gain for
                        one subblock */
    int reg0Cnt;         /* the number of scale factor bands in the
                        first region of <big_values> less one */
    int reg1Cnt;         /* the number of scale factor bands in the
                        second region of <big_values> less one */
    int preFlag;         /* flag indicating high frequency boost */
    int sfScale;         /* scale factor scaling */
    int cnt1TabSel;      /* Huffman table index for the <count1>
                        field of quadruples */
} IppMP3SideInfo;
```


MP3 Huffman Table Structure

```
typedef struct {
    int tableSize;          /* number of rows in table */
    int linbits;            /* variable used for encode if the magnitude of
                           encoded value is greater or equal to 15 */
    int maxBitsValue;       /* maximum bit length of codewords */
    Ipp16u *pHcod;          /* pointer to Huffman code table */
    Ipp8u *pSlen;           /* pointer to Huffman length table */
} IppMP3HuffmanTable;
```

MP3 Psychoacoustic Model Two Analysis

```
typedef struct{
    Ipp32s pMSR[36]; /* MSRs for one granule/channel.

        For long blocks, elements 0-20 represent the thresholds
        associated with the 21 SFBs. For short blocks, elements
        0,3,6,...,33, elements 1,4,...,34, and elements 2,5,...,35,
        respectively, represent the thresholds associated with the
        12 SFBs for each of the 3 consecutive short blocks in one
        granule/channel. That is, the block thresholds are
        interleaved such that the thresholds are grouped by SFB.*/

    Ipp32s PE;            /* Estimated perceptual entropy, one
                           granule/channel */
} IppMP3PsychoacousticModelTwoAnalysis;
```

Psychoacoustic Model Two State

```
typedef struct {
    Ipp64s pPrevMaskedThesholdLong[2][63]; /* long block masked
        threshold history buffer; contains masked
        threshold estimates for the threshold
        calculation partitions associated with the
        two most recent long blocks */

    Ipp64s pPrevMaskedThesholdShort[IPP_MP3_PSY_BAND_SHORT_NUM]; /*
        short block masked threshold history
        buffer; contains masked threshold
```

```
        estimates for the threshold calculation
        partitions associated with the two most
        recent short blocks */

Ipp32sc pPrevFFT[2][FIRST_6_CW]; /* FFT history buffer; contains
                                   real and imaginary FFT components
                                   associated with the two most recent
                                   long blocks */

Ipp32s  pPrevFTTMag[2][FIRST_6_CW]; /* FFT magnitude history buffer;
                                       contains FFT component magnitudes
                                       associated with the two most
                                       recent long blocks */

int nextPerceptualEntropy; /* PE estimate for next granule; one
                            granule delay provided for synchronization
                            with analysis filterbank */

int nextBlockType; /* Expected block type for next granule; either
                    long (normal), short, or stop. Depending upon
                    analysis results for the granule following the
                    next, a long block could change to a start block,
                    and a stop block could change to a short block.
                    This buffer provides one granule of delay for
                    synchronization with the analysis filterbank */

Ipp32s pNextMSRLong[21]; /* long block MSR estimates for next
                           granule. One granule delay provided for
                           synchronization with analysis filterbank */

Ipp32s pNextMSRShort[36]; /* short block MSR estimates for next
                            granule. One granule delay provided for
                            synchronization with analysis filterbank */
} IppMP3PsychoacousticModelTwoState;
```

MP3 Bit Reservoir

```
typedef struct {  
    int BitsRemaining;    /* bits currently remaining in the  
                           reservoir */  
  
    int MaxBits;          /* maximum possible reservoir size, in  
                           bits, determined as follows:  
                           min(7680-avg_frame_len, 2^9*8),  
                           where: avg_frame_len is the average frame  
                           length (in bits), including padding bits  
                           and excluding side information bits */  
  
} IppMP3BitReservoir;
```

MP3 Codec Enumerated Types

The Intel® IPP MP3 encoder and decoder APIs define enumerated data types that facilitate the synchronization and data transfer between the encoder and decoder components. As shown in [Table 10-3](#), the MP3 codec API includes several enumerated types that provide semantic interpretations for frequently used constants.

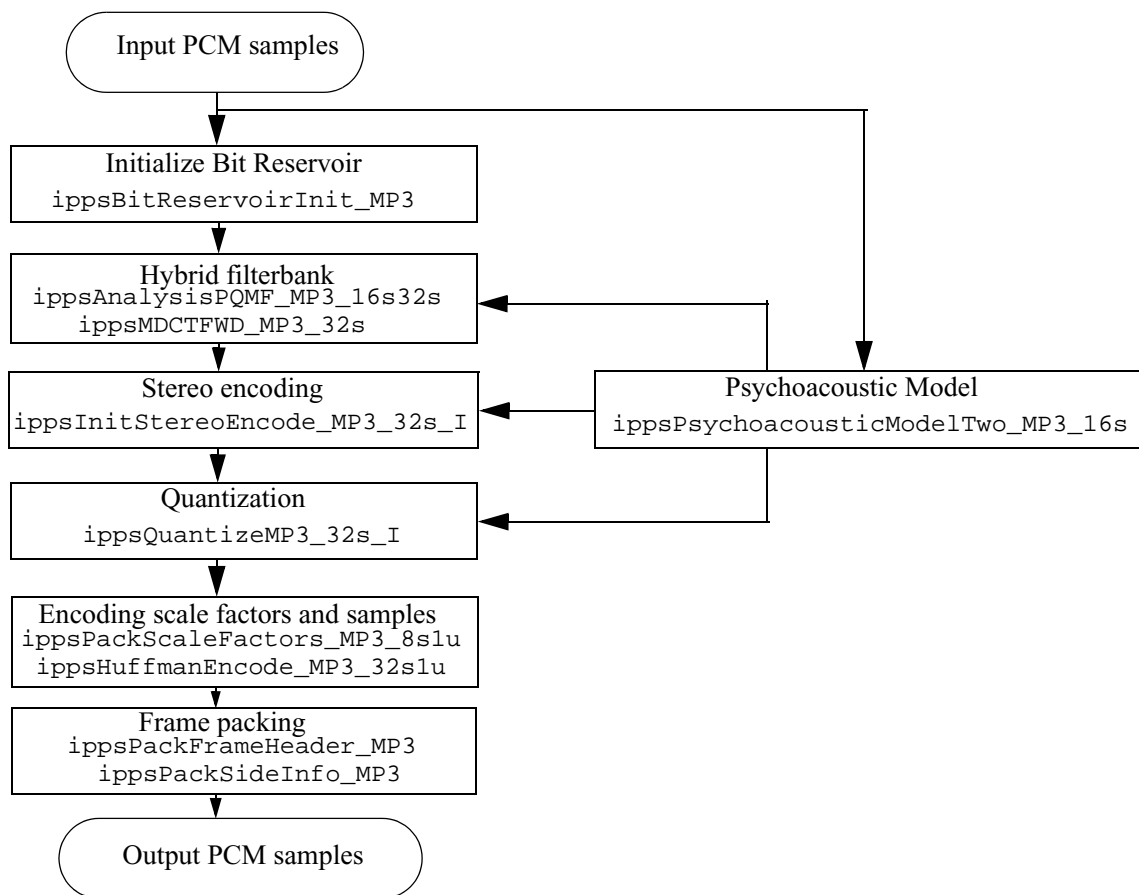
Table 10-3 MP3 Enumerated Data Types

Enumerated Type Name	Symbolic Values	Constant Value
IppMP3BitRate	ippMP3BitRateFree	0
	ippMP3BitRate32	1
	ippMP3BitRate40	2
	ippMP3BitRate48	3
	ippMP3BitRate56	4
	ippMP3BitRate64	5
	ippMP3BitRate80	6
	ippMP3BitRate96	7
	ippMP3BitRate112	8
	ippMP3BitRate128	9
	ippMP3BitRate160	10
	ippMP3BitRate192	11
	ippMP3BitRate224	12
	ippMP3BitRate256	13
	ippMP3BitRate320	14
IppMP3SampleRate	ippMP3SampleRate32000	2
	ippMP3SampleRate44100	0
	ippMP3SampleRate48000	1
IppMP3PcmMode	ippMP3NonInterleavedPCM	1
	ippMP3InterleavedPCM	2
IppMP3Emphasis	IppMP3EmphasisNone	0
	IppMP3Emphasis5015	1
	IppMP3EmphasisReserved	2
	IppMP3CCITTJ17	3

MP3 Audio Encoder

The MP3 encoder Application Programming Interface (API) provides a variety of capabilities, including bitstream packing functions and MP3 core encoding functions. See [ISO-11172](#). This chapter provides a reference guide to the Intel® Integrated Performance Primitives (Intel® IPP) MP3 audio encoder API. As shown in [Figure 10-1](#), this API includes several functions as well as predefined macros and constants.

Figure 10-1 Intel® IPP MP3 Encoder API Flowchart



AnalysisPQMF_MP3

Implements stage 1 of MP3 hybrid analysis filterbank.

Syntax

```
IppStatus ippsAnalysisPQMF_MP3_16s32s(const Ipp16s* pSrcPcm,  
    Ipp32s* pDstS, int pcmMode);
```

Parameters

- pSrcPcm* Pointer to the start of the buffer containing the input PCM audio vector. The samples conform to the following guidelines:
- must be in 16-bit, signed, little-endian, Q15 format
 - most recent 480 (512-32) samples should be contained in the vector $pSrcPcm[pcmMode*i]$, where $i = 0, 1, \dots, 479$
 - samples associated with the current granule should be contained in the vector $pSrcPcm[pcmMode*j]$, where $j = 480, 481, \dots, 1055$.
- pcmMode* PCM mode flag. Communicates to PQMF filterbank the type of input PCM vector organization to expect:
- $pcmMode = 1$ denotes non-interleaved PCM input samples
 - $pcmMode = 2$ denotes interleaved PCM input samples.
- pDstS* Pointer to the start of the 576-element block PQMF analysis output vector containing 18 consecutive blocks of 32 subband samples under the following index:
 $pDstXS[32*i + sb]$, where $i = 0, 1, \dots, 17$ is time series index
 $sb = 0, 1, \dots, 31$ is the subband index.

Description

The function is declared in the `ipps.h` file. This function implements the first stage of MP3 hybrid analysis filterbank. The function applies the critically sampled block PQMF analysis bank characterized by the 512-sample prototype window to a PCM input audio vector.

Call the `ippsAnalysisPQMF_MP3_16s32s` function 18 times per granule on each channel, that is, 36 times per channel on each frame.



NOTE. All coefficients are represented using the Q7.24 format

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcPcm</i> or <i>pDstXs</i> is NULL.
<code>ippStsErr</code>	Indicates an error when <i>pcmMode</i> exceeds [1, 2].

MDCTFwd_MP3

Implements stage 2 of the MP3 hybrid analysis filterbank.

Syntax

```
IppStatus ippMDCTFwd_MP3_32s(const Ipp32s* pSrc, Ipp32s* pDst, int blockType,
    int mixedBlock, IppMP3FrameHeader* pFrameHeader, Ipp32s* pOverlapBuf);
```

Parameters

<i>pSrc</i>	<p>Pointer to the start of the 576-element block PQMF analysis output vector containing 18 consecutive blocks of 32 subband samples that are indexed as follows:</p> <p>$pDstS[32*i+sb]$, where $i = 0, 1, \dots, 17$ is time series index, $sb = 0, 1, \dots, 31$ is the subband index.</p> <p>All coefficients are represented using the Q7.24 format.</p>
<i>blockType</i>	<p>Block type indicator:</p> <ul style="list-style-type: none"> 0 stands for normal block 1 stands for start block 2 stands for short block 3 stands for stop block.

<i>mixedBlock</i>	Mixed block indicator: 0 stands for not mixed 1 stands for mixed.
<i>pFrameHeader</i>	Pointer to the <code>IppMP3FrameHeader</code> structure that contains the header associated with the current frame. Only MPEG-1 (<i>id</i> = 1) is supported.
<i>pOverlapBuf</i>	Pointer to the MDCT overlap buffer that contains a copy of the most recent 576-element block of PQMF bank outputs. Prior to processing a new audio stream with the analysis filterbank, all elements of the buffer <i>pOverlapBuf</i> should be initialized to the constant value 0.
<i>pDst</i>	Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank.

Description

The function is declared in the `ipps.h` file. This function implements the second stage of MP3 hybrid analysis filterbank by performing the following operations:

1. **Forward MDCT.** An appropriately arranged set of 12-point and/or 36-point forward Modified Discrete Cosine Transforms (MDCTs) is applied to the 18-sample spectral coefficient blocks generated on each of the 32 PQMF subbands during the first stage analysis.
2. **Aliasing reduction butterflies.** The butterflies specified in [ISO/IEC 11172-3](#) are applied to the MDCT outputs in order to mitigate the aliasing artifacts introduced by cascading two critically sampled analysis filterbanks. Each of these introduces some non-negligible amount of interband aliasing.

The function `ippsMDCTFwd_MP3_32s` updates the 576-element MDCT overlap buffer *pMDCTOverlap*[], the contents of which must be preserved between calls to facilitate coherent block processing. The function must be applied once per granule on each channel (that is, applied twice per channel on each frame).



NOTE. Input coefficients are represented in the Q7.24 format. Output coefficients are represented in the Q5.26 format.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the pointers <code>pSrcXs</code> , <code>pDstXr</code> , <code>pFrameHeader</code> , or <code>pOverlapBuf</code> is NULL.

PsychoacousticModelTwo_MP3

Implements ISO/IEC 11172-3 psychoacoustic model recommendation 2 to estimate masked threshold and perceptual entropy associated with a block of PCM audio input.

Syntax

```
IppStatus ippPsychoacousticModelTwo_MP3_16s(const Ipp16s* pSrcPcm,
    IppMP3PsychoacousticModelTwoAnalysis* pDstPsyInfo, int* pDstIsSfbBound,
    IppMP3SideInfo* pDstSideInfo, IppMP3FrameHeader* pFrameHeader,
    IppMP3PsychoacousticModelTwoState* pFramePsyState, Ipp32s* pWorkBuffer,
    int pcmMode);
```

Parameters

<code>pSrcPcm</code>	<p>Pointer to the start of the buffer containing the input PCM audio vector, the samples of which should conform to the following format specification:</p> <ul style="list-style-type: none"> • 16-bits per sample, signed, little-endian, Q15. • <code>pSrcPcm</code> buffer should contain 1152 samples, that is, two granules of 576 samples each, if the parameter <code>pFrameHeader -> mode</code> has the value 1 (mono), or 2304 samples, that is, two granules of 576 samples each, if the parameter <code>pFrameHeader -> mode</code> has the value of 2 (stereo, dual mono). • In the stereophonic case, the PCM samples associated with the left and right channels should be organized according to the <code>pcmMode</code> flag. Failure to satisfy any of the above PCM format and/or buffer requirements results in undefined model outputs.
----------------------	---

<i>pDstPsychoInfo</i>	<p>Pointer to the first element in a set of <code>PsychoacousticModelTwoAnalysis</code> structures. Each set member contains the MSR and PE estimates for one granule. The number of elements in the set is equal to the number of channels, with the outputs arranged as follows:</p> <p>(Analysis[0] = granule 1, channel 1), (...Analysis[1] = granule 1, channel 2), (...Analysis[2] = granule 2, channel 1), (...Analysis[3] = granule 2, channel 2).</p>
<i>pDstIsSfbBound</i>	<p>If intensity coding has been enabled, <i>pDstIsSfbBound</i> points to the list of SFB lower bounds above which all spectral coefficients should be processed by the joint stereo intensity coding module. Since the intensity coding SFB lower bound is block-specific, the number of valid elements pointed to by <i>pDstIsSfbBound</i> varies depending upon the individual block types associated with each granule. In particular, the list of SFB bounds is indexed as follows:</p> <p><i>pIsSfbBound</i>[3*<i>gr</i>] for long block granules <i>pIsSfbBound</i>[3*<i>gr</i> + <i>w</i>] for short block granules,</p> <p>where <i>gr</i> is the granule index (0 indicates granule 1 and 1 indicates granule 2), and <i>w</i> is the block index (0 indicates block 1, 1 indicates block 2, 2 indicates block 3).</p> <p>For example, given short-block analysis in granule 1 followed by long block analysis in granule 2, the list of SFB bounds would be generated in the following order:</p> <p><i>pIsSfbBound</i>[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}.</p> <p>Only one SFB lower bound decision is generated for long block granules, whereas three are generated for short block granules. If both MS and intensity coding are enabled, then the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding has been enabled, then the SFB bound represents the lowest non-MS SFB.</p>
<i>pDstSideInfo</i>	<p>Pointer to the updated set of <code>IppMP3SideInfo</code> structures associated with all granules and channels. The model updates the following fields in all set elements: <i>blockType</i>, <i>winSwitch</i>, and <i>mixedBlock</i>. The number of elements in the set is equal to 2 times the number of channels. Ordering of the set elements is the same as <i>pDstPsychoInfo</i>.</p>

<i>pFrameHeader</i>	<p>Pointer to the updated <code>IppMP3FrameHeader</code> structure that contains the header associated with the current frame. The model updates the element <i>modeExt</i> to reflect the joint stereo coding mode decision. No other frame header fields are modified by this function.</p>
<i>pFramePsyState</i>	<p>Pointer to the first element in a set of <code>IppMP3PsychoacousticModelTwoState</code> structures that contains the updated psychoacoustic model state information associated with both the current frame and next frame. The number of elements in the set is equal to the number of channels contained in the input audio. That is, a separate analysis is carried for each channel.</p> <p>Prior to encoding a new audio stream, all elements of the psychoacoustic model state structure <i>pPsychoacousticModelState</i> should be initialized to contain the value 0.</p> <p>In the signal processing domain, this could be accomplished using the function <code>ippsZero_16s</code> as follows:</p> <pre>ippsZero_16s ((Ipp16s *) pPsychoacousticModelState, sizeof(IppMP3Psychoacoustic ModelTwoState)/sizeof(Ipp16s)).</pre>
<i>pWorkBuffer</i>	<p>Pointer to the workspace buffer internally used by the psychoacoustic model for storage of intermediate results and other temporary data. The buffer length must be at least 25,200 bytes, that is, 6300 elements of type <code>Ipp32s</code>.</p>
<i>pcmMode</i>	<p>PCM mode flag. Communicates the psychoacoustic model which type of PCM vector organization to expect:</p> <ul style="list-style-type: none"> • <i>pcmMode</i> = 1 denotes non-interleaved PCM input samples, that is, <i>pSrcPcm</i>[0..1151] contains the input samples associated with the left channel, and <i>pSrcPcm</i>[1152..2303] contains the input samples associated with the right channel. • <i>pcmMode</i> = 2 denotes interleaved PCM input samples, that is, <i>pSrcPcm</i>[2*i] and <i>pSrcPcm</i>[2*i+1] contain the samples associated with the left and right channels, respectively, where <i>i</i> = 0,1,...,1151. <p>You can also use appropriately typecast elements <code>ippMP3NonInterleavedPCM</code> and <code>ippMP3InterleavedPCM</code> of the enumerated type <code>IppMP3PcmMode</code> as an alternative to the constants 1 and 2 for <i>pcmMode</i>.</p>

Description

The function is declared in the `ipps.h` file. This function implements the [ISO/IEC 11172-3](#) psychoacoustic model recommendation 2 to estimate the masked threshold and perceptual entropy associated with a block of PCM audio input. Quantization process uses model outputs to estimate a perceptually optimal bit allocation for the spectral coefficients generated by the analysis filterbank. The psychoacoustic model also controls stereophonic MS/intensity mode selection and processing as well as analysis filterbank block size switching. Given one frame of PCM input audio of 1152 samples per channel, that is, two granules of 576 samples each, the psychoacoustic model generates the following outputs:

1. **Estimated SFB (scale factor band) Mask-to-Signal ratios (MSRs).** The model generates a vector of estimated MSRs for the 21 SFBs in long block mode and 12 SFBs for each of three consecutive blocks in short block mode.

The MSR is derived from the masked threshold, which quantifies the simultaneous masking power associated with one granule/channel (576 samples) of input audio. Given the properties of the audio stimulus presented to the listener, this threshold essentially quantifies the granule-instantaneous modified threshold of hearing. Ideally, the threshold estimate should provide a frequency-dependent intensity (dB SPL) profile beneath which an average listener cannot perceive quantization noise or, for that matter, any other spectral energy.

To estimate the masked threshold from a block of input audio, the function `ippsPsych_MP3_16s` implements the procedure recommended in Annex D.2 of [ISO/IEC 11172-3](#).

First, the output of a classical FFT-based spectral analysis is grouped into threshold calculation partitions that are organized to achieve analysis with sub-critical bandwidth resolution. On each threshold calculation partition, the model employs a weighted estimate of tone-like or noise-like signal behavior determined by an assessment of a spectral unpredictability across time to estimate masking power in each partition.

Second, a spreading function is applied to model the spectral selectivity of the auditory system.

Finally, the estimated threshold is compared against the absolute threshold of hearing in quiet and the maximum of the two is assigned to the threshold calculation partition. Ultimately, in order to match its output to the bit allocation scheme of the quantization module, the model converts from the threshold calculation partition scale to a scale factor band (SFB) scale. One set of 21 SFB thresholds is generated for long blocks (576

samples), or three consecutive blocks of 12 SFB thresholds are generated for short blocks (192 samples).

To facilitate efficient quantization, the SFB thresholds are inverted and normalized by the signal energy and returned in a vector of SFB Mask-to-Signal ratios (MSRs). The estimated MSRs are returned in the `PsychoacousticModelTwoAnalysis` structure.

2. **Estimated perceptual entropy.** The model generates a perceptual entropy (PE) estimate for each granule. The PE quantifies the minimum number of bits required to represent the PCM samples of the granule with “perceptual transparency”. That is, without audible loss of quality for an average listener in comparison to the original, uncoded version.

The estimated PE is derived from the masked threshold, in combination with classical assumptions about the minimum number of bits required to achieve a particular signal-to-noise ratio (SNR) target in each SFB, incremental per bit SNR improvement = +6 dB, where the minimum required SNR and hence minimum required number of bits for each SFB is derived from the signal-to-mask ratio (SMR).

Perceptual entropy is used to control analysis filterbank block size switching, since sudden large PE increases are often associated with transient audio events that are prone to pre-echo distortion. The PE estimate is returned in the `PsychoacousticModelTwoAnalysis` structure.

3. **Analysis filterbank block size decision.** Using perceptual entropy and other indicators, the model determines whether or not the current granule is susceptible to pre-echo distortion. You should prefer using the short block mode when pre-echoes are likely and use long blocks in all other cases.

In order to ensure selection of the appropriate block type, the decision incorporates single block look ahead switching logic. For example, if the current block type is long and the next block type is short, the current block type is changed from block type “long/normal” to block type “long/start” in order to guarantee seamless block processing upon mode switch. Similarly, if the current block type has been designated as “long/stop” and the next block type is determined to be “short”, the block switching logic changes the current block from “long/stop” back to “short” in order to avoid unnecessary mode switching. The block type decision is returned in the frame/granule `IppMP3SideInfo` structure.

4. **Joint stereophonic processing mode decision.** For 2-channel audio sources, the model evaluates interchannel correlations and other indicators in order to generate joint stereo LR/MS and/or intensity processing mode decisions. The joint stereo mode decision is returned in the `modeExt` field of the `IppMP3FrameHeader` structure.

5. **Intensity stereo coding SFB bound decision.** If intensity coding has been activated (see the preceding item joint stereophonic processing mode decision), the psychoacoustic model determines an appropriate lower SFB bound above which all spectral coefficients should be encoded using intensity mode stereophonic processing.

The psychoacoustic model performs analysis on a frame basis (1152 samples per channel), including two granules and up to two channels for either stereophonic or dual mono inputs. Valid lengths for both input and output vectors depend upon which mono or stereo channel modes have been enabled.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <code>pSrcPcm</code> , <code>pDstPsyInfo</code> , <code>pDstSideInfo</code> , <code>pDstIsSfbBound</code> , <code>pFrameHeader</code> , <code>pDstPsyState</code> , or <code>pWorkBuffer</code> is NULL.

JointStereoEncode_MP3

Transforms independent left and right channel spectral coefficient vectors into combined mid/side and/or intensity mode coefficient vectors suitable for quantization.

Syntax

```
IppStatus ippJointStereoEncode_MP3_32s_I(Ipp32s* pSrcDstXrL, Ipp32s*
    pSrcDstXrR, Ipp8s* pDstScaleFactorR, IppMP3FrameHeader* pFrameHeader,
    IppMP3SideInfo* pSideInfo, int* pIsSfbBound);
```

Parameters

<code>pSrcDstXrL</code>	Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank for the left channel of input audio. All coefficients are represented using the Q5.26 format.
<code>pSrcDstXrR</code>	Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank for the right channel of input audio. All coefficients are represented using the Q5.26 format.

<i>pFrameHeader</i>	<p>Pointer to the <code>IppMP3FrameHeader</code> structure that contains the header information associated with the current frame. Upon function entry, the structure fields <code>samplingFreq</code>, <code>id</code>, <code>mode</code>, and <code>modeExt</code> should contain, respectively, the sample rate associated with the current input audio, the algorithm <i>id</i> (MPEG-1 or MPEG-2), and the joint stereo coding commands generated by the psychoacoustic model. All other <i>pFrameHeader</i> fields are ignored.</p> <p>Only MPEG-1 (<code>id = 1</code>) is supported.</p>
<i>pSideInfo</i>	<p>Pointer to the pair of <code>IppMP3SideInfo</code> structures associated with the channel pair to be jointly encoded. The number of elements in the set is 2, and ordering of the set elements is as follows: <i>pSideInfo</i>[0] describes channel 1, and <i>pSideInfo</i>[1] describes channel 2. Upon the function entry, the <code>blockType</code> side information fields for both channels should reflect the analysis modes (short or long block) selected by the psychoacoustic model on each channel. All other fields in the <i>pSideInfo</i>[0] and <i>pSideInfo</i>[1] structures are ignored.</p>
<i>pIsSfbBound</i>	<p>Pointer to the list of intensity coding SFB lower bounds for both channels of the current granule above which all L/R channel spectral coefficients are combined into an intensity-coded representation. The number of elements depends on the block type associated with the current granule. For short blocks, the SFB bounds are represented in the following order: <i>pIsSfbBound</i>[0] describes block 1, <i>pIsSfbBound</i>[1] describes block 2, and <i>pIsSfbBound</i>[2] describes block 3.</p> <p>For long blocks, only a single SFB lower bound decision is required and is represented in <i>pIsSfbBound</i>[0]. If both MS and intensity coding are enabled, the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding is enabled, the SFB bound represents the lowest non-MS SFB.</p>
<i>pDstScaleFactorR</i>	<p>Pointer to the vector of scalefactors associated with one granule of the right/S channel. If intensity coding has been enabled by the psychoacoustic model above a certain SFB lower bound, as indicated by the frame header and the vector pointed to by <i>pIsSfbBound</i>, the function <code>StereoEncode_MP3_32s_I</code> updates with the appropriate scalefactors those elements of <i>pDstScaleFactorR</i>[] that are associated with intensity coded scale factor bands. Other SFB entries in the scale factor vector remain unmodified. The length of the vector</p>

referenced by *pDstScaleFactorR* varies as a function of block size. The vector contains 21 elements for long block granules, or 36 elements for short block granules.

Description

The function is declared in the `ipps.h` file. This function transforms the independent left and right channel spectral coefficient vectors into combined mid/side (MS) and/or intensity (IS) mode coefficient vectors suitable for quantization. If MS coding is enabled (`pFrameHeader -> modeExt & 0x10 == 1`), the left and right channels are converted to Mid and Side channels as follows:

$$M = \frac{L+R}{\sqrt{2}} \quad S = \frac{L-R}{\sqrt{2}}$$

This function is called on dual granule basis. You should call it for every granule/2 channels.

If intensity coding is enabled (`pFrameHeader -> modeExt & 0x01 = 1`), the left channel carries the intensity data for SFBs above the SFB intensity lower bound and the right channel above the SFB lower bound is cleared. All coefficients are set to 0.

$$L = L+R, R = 0$$

In order to facilitate energy-proportional recovery of the left and right spectral coefficients at the decoder, an intensity energy scale factor, *is_pos*, is transmitted in place of the right channel scale factor since the right channel spectral coefficients above the SFB bound are eliminated. The energy normalization constant is derived from the L/R SFB energy ratios and then transformed to improve its quantization properties. That is,

$$is_pos = n \operatorname{int} \left(\frac{12}{\pi} \arctan \left(\sqrt{\frac{L_energy}{R_energy}} \right) \right)$$

where *L_energy* and *R_energy* are, respectively, the SFB energies associated with the spectral coefficients of the left and right channels. At the decoder, the *is_pos* scale factor is used to apportion jointly coded signal energy between the left and right channels in a manner consistent with the distribution of signal energy prior to joint coding. The *is_pos* intensity scalefactors are returned in the vector *pDstScalefactorR*.

On each granule, joint stereo coding is applied once per channel pair (576 samples per granule on each channel). The function `JointStereoEncode_MP3_32s_I` must therefore be called once per granule, or twice per frame.

Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <code>pSrcDstXrL</code> , <code>pSrcDstXrR</code> , <code>pDstScaleFactorR</code> , <code>pFrameHeader</code> , <code>pSideInfo</code> , or <code>pIsSfbBound</code> is NULL.
<code>ippStsMP3SideInfoErr</code>	Indicates an error if <code>pSideInfo[0].blockType != pSideInfo[1].blockType</code> when IS or MS is used.

Quantize_MP3

Quantizes spectral coefficients generated by analysis filterbank.

Syntax

```
IppStatus ippsQuantize_MP3_32s_I(Ipp32s* pSrcDstXrIx, Ipp8s* pDstScalefactor,
int* pDstScfsi, int* pDstCount1Len, int* pDstHufSize, IppMP3FrameHeader*
pFrameHeader, IppMP3SideInfo* pSideInfo,
IppMP3PsychoacousticModelTwoAnalysis* pPsychoInfo,
IppMP3PsychoacousticModelTwoState* pFramePsyState, IppMP3BitReservoir*
pResv, int meanBits, int* pIsSfbBound, Ipp32s* pWorkBuffer);
```

Parameters

<code>pSrcDstXrIx</code>	<p>Pointer to the set of unquantized spectral coefficient vectors generated by the analysis filterbank and optionally processed by the joint stereo coding module for one frame. The set of unquantized coefficients should be indexed as follows:</p> <ul style="list-style-type: none"> • <code>pSrcDstXrIx[gr*1152 + ch*576 + i]</code> for stereophonic and dual-mono input sources • <code>pSrcDstXrIx[gr*576 + i]</code> for monaural, that is, input sources, <p>where</p> <ul style="list-style-type: none"> • <code>i = 0,1,...,575</code> is the spectral coefficient index • <code>gr</code> is the granule index. 0 stands for granule 1, 1 stands for granule 2 • <code>ch</code> is the channel index. 0 stands for channel 1, 1 stands for channel 2.
--------------------------	---

Depending on which type of joint coding has been applied, if any, the coefficients for each channel could be associated with L/R, M/S, and/or intensity representations of the input audio. All coefficients should be represented using the Q5.26 format.

pDstScaleFactor

Pointer to the output set of scalefactors generated during the quantization process. These scalefactors determine the quantizer granularity. Scale factor vector lengths depend on the block mode associated with each granule. The order of the elements is:

1. (granule 1, channel 1)
2. (granule 1, channel 2)
3. (granule 2, channel 1)
4. (granule 2, channel 2).

Given this general organization, the side information for each granule/channel in conjunction with the flags contained in the vector *pDstScfsi* can be used to determine the precise scale factor vector indices and lengths.

pDstScfsi

Pointer to the output vector of scale factor selection information. This vector contains a set of binary flags that indicate whether or not scalefactors are shared across granules of a frame within predefined scale factor selection groups. For example, bands 0,1,2,3,4,5 form one group; bands 6,7,8,9,10 form a second group, as defined in [ISO/IEC 11172-3](#). The vector is indexed as follows:

pDstScfsi[ch][scfsi_band], where

- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.
- *scfsi_band* is the scale factor selection group number:
 - group 0 includes SFBs 0-5,
 - group 1 includes SFBs 6-10,
 - group 2 includes SFBs 11-15,
 - group 3 includes SFBs 16-20.

pDstCount1Len

Pointer to an output vector of *count1* region length specifiers. For the purposes of Huffman coding spectral coefficients of a higher frequency than the *bigvals* region, the *count1* parameter indicates the size of the region in which spectral samples can be combined into quadruples for

which all elements are of magnitude less than or equal to 1. The vector contains $2*nchan$, elements and is indexed as follows:

$pDstCount1Len[gr*nchan + ch]$, where

- gr is the granule index. 0 stands for granule 1, 1 stands for granule 2.
- $nchan$ is the number of channels.
- ch is the channel index. 0 stands for channel 1, 1 stands for channel 2.

pDstHufSize

Pointer to an output vector of Huffman coding bit allocation specifiers. For each granule/channel, the specifiers indicate the total number of Huffman bits required to represent the quantized spectral coefficients in the *bigvals* and *count1* regions.

Whenever necessary, each *HufSize* bit count is augmented to include the number of bits required to manage the bit reservoir. For frames in which the reservoir has reached the maximum capacity, the quantizer expends the surplus bits by padding with additional bits the Huffman representation of the spectral samples. The *HufSize* result returned by the quantizer reflects these padding requirements. That is, $HufSize[i]$ is equal to the sum of the number of bits required for Huffman symbols and the number of padding bits. The vector contains $2*nchan$, elements and is indexed as follows: $pDstHufSize[gr*nchan+ch]$, where

- gr is the granule index. 0 stands for granule 1, 1 stands for granule 2.
- $nchan$ is the number of channels.
- ch is the channel index. 0 stands for channel 1, 1 stands for channel 2.

pFrameHeader

Pointer to the *IppMP3FrameHeader* structure that contains the header information associated with the current frame. Upon the function entry, the structure fields *samplingFreq*, *id*, *mode*, and *modeExt* should contain, respectively, the sample rate associated with the current input audio, the algorithm *id*, that is, MPEG-1 or MPEG-2, and the joint stereo coding commands generated by the psychoacoustic model. All other **pFrameHeader* fields are ignored. Only MPEG-1 (*id* = 1) is supported.

<i>pSideInfo</i>	<p>Pointer to the set of <code>IppMP3SideInfo</code> structures associated with all granules and channels. The set should contain $2*nchan$, elements and should be indexed as follows: <code>pSideInfo[gr*nchan + ch]</code>, where</p> <ul style="list-style-type: none"> • <i>gr</i> is the granule index. 0 stands for granule 1, 1 stands for granule 2. • <i>nchan</i> is the number of channels. • <i>ch</i> is the channel index. 0 stands for channel 1, 1 stands for channel 2. <p>Upon the function entry, in all set elements the structure fields <i>blockType</i>, <i>mixedBlock</i>, and <i>winSwitch</i> should contain, respectively, the block type indicator (<i>start</i>, <i>short</i>, or <i>stop</i>), filter bank mixed block analysis mode specifier, and window switching flags (<i>normal</i> or <i>blockType</i>) associated with the current input audio. All other <i>*pSideInfo</i> fields are ignored upon the function entry and updated upon function exit, as described below under the description of output arguments.</p>
<i>pPsychoInfo</i>	<p>Pointer to the first element in a set of <code>PsychoacousticModelTwoAnalysis</code> structures associated with the current frame. Each set member contains the MSR and PE estimates for one channel of one granule. The set should contain $2*nchan$, elements and is indexed as: <code>pPsychoaInfo[gr*nchan+ ch]</code>, where</p> <ul style="list-style-type: none"> • <i>gr</i> is the granule index. 0 stands for granule 1, 1 stands for granule 2 • <i>nchan</i> is the number of channels • <i>ch</i> is the channel index. 0 stands for channel 1, 1 stands for channel 2.
<i>pFramePsyState</i>	<p>Pointer to the first element in a set of <code>IppMP3PsychoacousticModelTwoState</code> structures that contains the psychoacoustic model state information associated with both the current frame and next frame. The number of elements in the set is equal to the number of channels contained in the input audio. That is, a separate analysis is carried for each channel. The quantizer uses the frame type look ahead information <i>nextBlockType</i> to manage the bit reservoir. All other structure elements are ignored by the quantizer.</p>
<i>pResv</i>	<p>Pointer to the <code>IppMP3BitReservoir</code> structure that contains the bit reservoir state information. Upon the function entry, all structure fields should contain valid data.</p>

meanBits

The number of bits allocated on an average basis for each frame of spectral coefficients and scalefactors given the target bit rate (kilobits per second) specified in the frame header. This number excludes the bits allocated for the frame header and side information. The quantizer uses *meanBits* as a target allocation for the current frame. Given perceptual bit allocation requirements greater than this target, the quantizer makes use of the surplus bits held in the bit reservoir to satisfy frame-instantaneous demands. Similarly, given perceptual bit allocation requirements below this target, the quantizer will store surplus bits in the bit reservoir for use by future frames.

pIsSfbBound

Pointer to the list of SFB lower bounds above which all L/R channel spectral coefficients have been combined into an intensity-coded representation. The number of valid elements pointed to by *pIsSfbBound* depends upon the block types associated with the granules of the current frame.

In particular, the list of SFB bounds pointed to by *pIsSfbBound* is indexed as follows: *pIsSfbBound[3*gr]* for long block granules, and *pIsSfbBound[3*gr + w]* for short block granules, where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2
- *w* is the block index. 0 stands for block 1, 1 stands for block 2, 2 stands for block 3.

For example, given short-block analysis in granule 1 followed by long block analysis in granule 2, the list of SFB bounds would be expected in the following order:

pIsSfbBound[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}.

If a granule is configured for long block analysis, then only a single SFB lower bound decision is expected, whereas three are expected for short block granules. If both MS and intensity coding have been enabled, then the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding has been enabled, then the SFB bound represents the lowest non-MS SFB.

pWorkBuffer

Pointer to the workspace buffer internally used by the quantizer for storage of intermediate results and other temporary data. The buffer length should be at least 2880 bytes, that is, 720 words of 32-bits each.

Description

The function is declared in the `ipp.h` file. This function quantizes the spectral coefficients generated by the analysis filterbank such that the resulting distortion, that is, quantization noise, is shaped to match a profile derived from the masked thresholds estimated by the psychoacoustic model.

Along with meeting perceptual distortion criteria, the quantizer simultaneously adjusts the overall bit allocation to achieve a fixed bit rate target. In accordance with the [ISO/IEC 11172-3](#) recommendation, a bit reservoir is maintained in order to meet instantaneous peak rate demands without violating on an average basis the fixed rate constraint. Surplus bits are deposited in the reservoir during frames with lower than average perceptual bit rate requirements. Supplementary bits are withdrawn from the reservoir to satisfy frames with higher-than-average perceptual bit allocation requirements.

The quantizer manages the reservoir and overall bit allocation such that a constant average rate constraint is satisfied. The quantizer operates on a complete frame of data, that is, two granules and either one or two channels, and it should therefore be called once per frame.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <code>pSrcDstXrIx</code> , <code>pDstScaleFactor</code> , <code>pDstScfsi</code> , <code>pDstCount1Len</code> , <code>pDstHufSize</code> , <code>pFrameHeader</code> , <code>pSideInfo</code> , <code>pPsychInfo</code> , <code>pFramePsyState</code> , <code>pResv</code> , <code>pIsSfbBound</code> , or <code>pWorkBuffer</code> is NULL.
<code>ippStsMP3SideInfo</code>	Indicates an error when <code>pSideInfo->winSwitch</code> and <code>pSideInfo->mixedBlock</code> are both defined.
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <code>pFrameHeader->samplingFreq</code> or <code>pFrameHeader->id</code> is not correct.

PackScalefactors_MP3

Applies noiseless coding to scalefactors and packs output into bitstream buffer.

Syntax

```
IppStatus ippsPackScalefactors_MP3_8slu(const Ipp8s* pSrcScalefactor, Ipp8u**
    ppBitStream, int* pOffset, IppMP3FrameHeader* pFrameHeader, IppMP3SideInfo*
    pSideInfo, int* pScfsi, int granule, int channel);
```

Parameters

<i>pSrcScaleFactor</i>	<p>Pointer to a vector of scalefactors generated during the quantization process for one channel of one granule. Scale factor vector lengths depend on the block mode. Short block granule scale factor vectors contain 36 elements, or 12 elements for each subblock. Long block granule scale factor vectors contain 21 elements.</p> <p>Thus short block scale factor vectors are indexed as follows: <i>pSrcScaleFactor[sb*12+sfb]</i>, where</p> <ul style="list-style-type: none"> <i>sb</i> is the subblock index. 0 stands for subblock 1, 1 stands for subblock 2, 2 stands for subblock 3. <i>sfb</i> is the scale factor band index (0-11). <p>Long block scale factor vectors are indexed as follows: <i>pSrcScaleFactor[sfb]</i>, where <i>sfb</i> is the scale factor band index (0-20).</p> <p>The associated side information for an individual granule/channel can be used to select the appropriate indexing scheme.</p>
<i>ppBitStream</i>	<p>Updated bitstream byte pointer. This parameter points to the first available bitstream buffer byte immediately following the bits generated by the scale factor Huffman encoder and sequentially written into the stream buffer. The scale factor bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3.</p>

<i>pOffset</i>	Updated bitstream bit pointer. The <i>pOffset</i> parameter indexes the next available bit in the next available byte referenced by the updated bitstream buffer byte pointer <i>ppBitStream</i> . This parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit.
<i>pFrameHeader</i>	Pointer to the <i>IppMP3FrameHeader</i> structure for this frame. Upon the function entry, the structure fields <i>id</i> and <i>modeExt</i> should contain, respectively, the algorithm <i>id</i> (MPEG-1 or MPEG-2) and the joint stereo coding commands generated by the psychoacoustic model. All other <i>pFrameHeader</i> fields are ignored. Only MPEG-1 (<i>id</i> = 1) is supported.
<i>pSideInfo</i>	Pointer to the <i>IppMP3SideInfo</i> structure for the current granule and channel. Upon function entry, the structure fields <i>blockType</i> , <i>mixedBlock</i> , and <i>sfCompress</i> should contain, respectively, the block type indicator <i>start</i> , <i>short</i> , or <i>stop</i> , filter bank mixed block analysis mode specifier, and scale factor bit allocation. All other <i>*pSideInfo</i> fields are ignored by the scale factor encoder.
<i>pScfsi</i>	<p>Pointer to the scale factor selection information table that contains the set of binary flags that indicate whether scalefactors are shared across granules of a frame within the predefined scale factor selection groups.</p> <p>For example, bands 0,1,2,3,4,5 form one group and bands 6,7,8,9,10 form a second group (as defined in ISO/IEC 11172-3). The vector is indexed as follows: <i>pScfsi[ch][scfsi_band]</i>, where</p> <ul style="list-style-type: none"> • <i>ch</i> is the channel index. 0 stands for channel 1, 1 stands for channel 2. • <i>scfsi_band</i> is the scale factor selection group number: <ul style="list-style-type: none"> group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, group 3 includes SFBs 16-20.
<i>granule</i>	Index of the current granule. 0 stands for granule 1, 1 stands for granule 2.
<i>channel</i>	Index of the current channel. 0 stands for channel 1, 1 stands for channel 2.

Description

The function is declared in the `ipp.h` file. This function applies noiseless coding to the scalefactors and then packs the output into the bitstream buffer. This function operates on one channel of one granule at a time. Therefore, you should call this function once for each channel of each granule.

The resulting bitstream is fully compliant with the syntax specified in [ISO/IEC 11172-3](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsMP3SideInfoErr</code>	Indicates an error when <i>pFrameHeader->id</i> == IPP_MP3_ID_MPEG1 and <i>pSideInfo->sfCompress</i> exceeds [0..15]; <i>pFrameHeader->id</i> == IPP_MP3_ID_MPEG2 and <i>pSideInfo->sfCompress</i> exceeds [0..511].
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <i>pFrameHeader->id</i> == IPP_MP3_ID_MPEG2 and <i>pFrameHeader->modeExt</i> exceeds [0..3].

HuffmanEncode_MP3

Applies lossless Huffman encoding to quantized samples and packs output into bitstream buffer.

Syntax

```
IppStatus ippHuffmanEncode_MP3_32slu(Ipp32s* pSrcIx, Ipp8u** ppBitStream, int*
    pOffset, IppMP3FrameHeader* pFrameHeader, IppMP3SideInfo* pSideInfo, int
    count1Len, int hufSize);
```

Parameters

<i>pSrcIx</i>	Pointer to the quantized samples of a granule. The buffer length is 576. Depending on which type of joint coding has been applied, if any, the coefficient vector might be associated with either the L, R, M, S, and/or intensity channel of the quantized spectral data.
<i>ppBitStream</i>	Updated bitstream byte pointer. The parameter <i>*ppBitStream</i> points to the first available bitstream buffer byte immediately following the bits generated by the spectral coefficient Huffman encoder and sequentially written into the stream buffer. The Huffman symbol bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3 .
<i>pOffset</i>	Updated bitstream bit pointer. The <i>pOffset</i> parameter indexes the next available bit in the next available byte referenced by the updated bitstream buffer byte pointer <i>ppBitStream</i> . This parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit.
<i>pFrameHeader</i>	Pointer to the <code>IppMP3FrameHeader</code> structure for this frame. The Huffman encoder uses the frame header id field in connection with the side information (as described below) to compute the Huffman table region boundaries for the big value spectral region. The Huffman encoder ignores all other frame header fields. Only MPEG-1 (<i>id</i> = 1) is supported.
<i>pSideInfo</i>	<p>Pointer to the <code>IppMP3SideInfo</code> structure for the current granule and channel. The structure elements <code>bigVals</code>, <code>pTableSelect[0]-[2]</code>, <code>reg0Cnt</code>, and <code>reg1Cnt</code> are used to control coding of spectral coefficients in the big value region. The structure element <code>cnt1TabSel</code> is used to select the appropriate Huffman table for the (-1,0,+1)-valued 4-tuples in the <i>count1</i> region.</p> <p>For detailed descriptions of all side information elements, see the structure definition header file.</p>
<i>count1Len</i>	The <i>count1</i> region length specifier. Indicates the number of spectral samples for the current granule/channel above the big value region that can be combined into 4-tuples in which all elements are of magnitude less than or equal to 1.
<i>hufSize</i>	Huffman coding bit allocation specifier. Indicates the total number of bits that are required to represent the Huffman-encoded quantized spectral coefficients for the current granule/channel in both the <i>bigvals</i> and <i>count1</i> regions.

Whenever necessary, this bit count should be augmented to include the number of bits required to manage the bit reservoir. For frames in which the reservoir has reached maximum capacity, the surplus bits are expended by padding with additional bits the Huffman representation of the spectral samples.

The *HufSize* result returned by the function `Quantize_MP3_32s_I` reflects these padding requirements. That is, *HufSize*[*i*] is equal to the total of the number of bits required for Huffman symbols and the number of padding bits.

Description

The function is declared in the `ipp.h` file. This function applies lossless Huffman encoding to the quantized samples and packs the output into the bitstream buffer.

The function encodes one granule at a time, and therefore must be called once for each granule of each channel.

The resulting bitstream is fully compliant with [ISO/IEC 11172-3](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>pOffset</i> exceeds [0,7].
<code>ippStsMP3SideInfoErr</code>	Indicates an error when <code>pSideInfo->bigVals*2 > IPP_MP3_GRANULE_LEN</code> <code>(pSideInfo->reg0Cnt + pSideInfo->reg1Cnt + 2) >= 23,</code> <code>pSideInfo->cnt1TabSel</code> exceeds [0,1], <code>pSideInfo->pTableSelect[i]</code> exceeds [0..31].
<code>ippStsMP3FrameHeader</code>	Indicates an error when <code>pFrameHeader->id != IPP_MP3_ID_MPEG1</code> <code>pFrameHeader->layer != 1</code> <code>pFrameHeader->samplingFreq</code> exceeds [0..2].

PackFrameHeader_MP3

Packs the content of the frame header into the bitstream.

```
IppStatus ippSPackFrameHeader_MP3 (IppMP3FrameHeader* pSrcFrameHeader, Ipp8u**  
    ppBitStream);
```

Parameters

<i>pSrcFrameHeader</i>	Pointer to the <code>IppMP3FrameHeader</code> structure. This structure contains all the header information associated with the current frame. All structure fields must contain valid data upon function entry.
<i>ppBitStream</i>	Updated bitstream byte pointer. The parameter <i>ppBitStream</i> points to the first available bitstream buffer byte immediately following the packed frame header bits. The frame header bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3 .

Description

The function is declared in the `ippac.h` file. This function packs the content of the frame header into the bitstream.

The resulting bitstream is fully compliant with the syntax specified in [ISO/IEC 11172-3](#).

The function should be called once per frame.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcFrameHeader</i> or <i>ppBitStream</i> is <code>NULL</code> .

PackSideInfo_MP3

Packs the side information into the bitstream buffer.

Syntax

```
IppStatus ippsPackSideInfo_MP3(IppMP3SideInfo* pSrcSideInfo, Ipp8u** ppBitStream,
    int mainDataBegin, int privateBits, int* pSrcScfsi, IppMP3FrameHeader*
    pFrameHeader);
```

Parameters

<i>pSrcSideInfo</i>	<p>Pointer to the <code>IppMP3SideInfo</code> structures. This should contain twice the channel number of elements. The order is the following:</p> <ul style="list-style-type: none"> • granule 1, channel 1 • granule 1, channel 2 • granule 2, channel 1 • granule 2, channel 2. <p>All fields of all set elements should contain valid data upon the function entry.</p>
<i>ppBitStream</i>	<p>Updated bitstream byte pointer. The parameter <i>*ppBitStream</i> points to the first available bitstream buffer byte immediately following the packed side information bits. The frame header bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3:1993.</p>
<i>mainDataBegin</i>	<p>Negative bitstream offset, in bytes. The parameter value is typically the number of bytes remaining in the bit reservoir before the start of quantization for the current frame. When computing <i>mainDataBegin</i>, you should exclude the header and side information bytes.</p> <p>The side information formatter packs the 9-bit value of <i>mainDataBegin</i> into the <code>main_data_begin</code> field of the output bitstream.</p>
<i>privateBits</i>	<p>Depending on the number of channels, the function extracts the appropriate number of least significant bits from the parameter <i>privateBits</i> and packs them into the <code>private_bits</code> field of the output bitstream. The ISO/IEC 11172-3 bitstream syntax reserves a channel-dependent number of application-specific (private) bits in the</p>

layer III bitstream audio data section immediately following the parameter `main_data_begin`. See [ISO/IEC 11172-3:1993](#). For dual- and single-channel streams, respectively, three and five bits are reserved.

pSrcScfsi

Pointer to the scale factor selection information table. This vector contains a set of binary flags that indicate whether scalefactors are shared across granules of a frame within predefined scale factor selection groups. For example, bands 0,1,2,3,4,5 form one group and bands 6,7,8,9,10 form the second group, as defined in [ISO/IEC 11172-3 \[2\]](#).

The vector is indexed as `pDstScfsi[ch][scfsi_band]`, where

- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.
- *scfsi_band* is the scale factor selection group number:
 - group 0 includes SFBs 0-5,
 - group 1 includes SFBs 6-10,
 - group 2 includes SFBs 11-15,
 - group 3 includes SFBs 16-20.

pFrameHeader

Pointer to the `IppMP3FrameHeader` structure. Only MPEG-1 (*id* = 1) is supported. Upon the function entry, the structure fields *id*, *mode*, and *layer* should contain, respectively, the algorithm *id* (MPEG-1 or MPEG-2), the mono or stereo mode, and the MPEG layer specifier.

All other *pFrameHeader* fields are ignored.

Description

The function is declared in the `ippac.h` file. This function packs the side information into the bitstream buffer.

The resulting bitstream is fully compliant with the syntax specified in [ISO/IEC 11172-3](#).

This function should be called once per frame.

Return Values

`ippStsNoErr`

Indicates no error.

`ippStsNullPtrErr`

Indicates an error when *pSrcSideInfo*, *ppBitStream*, *pSrcScfsi*, or *pFrameHeader* is NULL.

<code>ippStsMP3FrameHeaderErr</code>	<p>Indicates an error when</p> <pre> pFrameHeader->id != IPP_MP3_ID_MPEG1 pFrameHeader->id != IPP_MP3_ID_MPEG2 pFrameHeader->layer != 1 pFrameHeader->mode exceeds [0..3]. </pre>
--------------------------------------	---

BitReservoirInit_MP3

Initializes all elements of the bit reservoir state structure.

Syntax

```

IppStatus ippSBitReservoirInit_MP3(IppMP3BitReservoir* pDstBitResv,
    IppMP3FrameHeader* pFrameHeader);

```

Parameters

<code>pFrameHeader</code>	<p>Pointer to the <code>IppMP3FrameHeader</code> structure that contains the header information associated with the current frame. The frame header fields <code>bitRate</code> and <code>id</code>, bit rate index and algorithm identification, respectively, must contain valid data prior to calling the function <code>BitReservoirInit_MP3</code> since both are used to generate the bit reservoir initialization parameters.</p> <p>All other frame header parameters are ignored by the bit reservoir initialization function. Only MPEG-1 (<code>id = 1</code>) is supported.</p>
<code>pDstBitResv</code>	<p>Pointer to the initialized <code>IppMP3BitReservoir</code> state structure. The structure element <code>BitsRemaining</code> is initialized as 0. The structure element <code>MaxBits</code> is initialized to reflect the maximum number of bits that can be contained in the reservoir at the start of any given frame. The appropriate value of <code>MaxBits</code> is directly determined by the selected algorithm (MPEG-1 or MPEG-2) and the stream bit rate indicated by the rate index parameter <code>pFrameHeader.bitRate</code>.</p>

Description

The function is declared in the `ippac.h` file. This function initializes all elements of the bit reservoir state structure based on the coding algorithm (MPEG-1 or MPEG-2) and the average per frame bit allocation specified in the frame header.

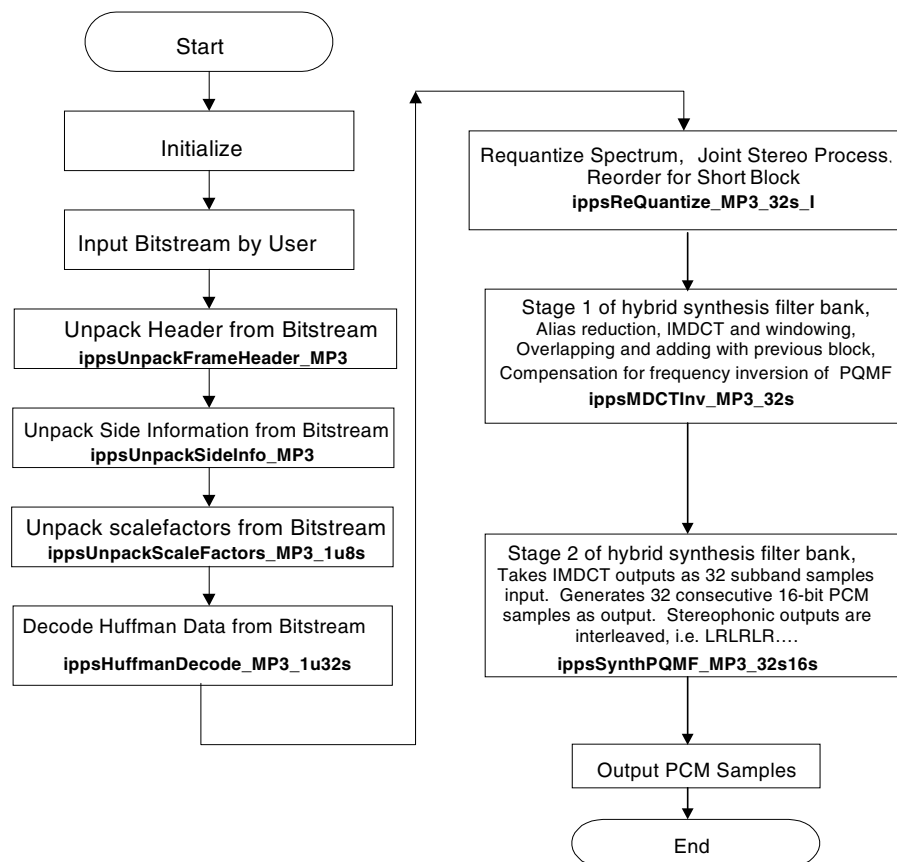
Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDstBitResv</i> or <i>pFrameHeader</i> is NULL.
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <i>pFrameHeader->id</i> != IPP_MP3_ID_MPEG1.

MP3 Audio Decoder

This section describes Intel IPP functions for assembling audio decoders compliant with the layer III portions of the ISO/IEC 11172-3 MPEG-1 and ISO/IEC 13818-3 MPEG-2 audio specifications (see [ISO]) and provides a reference guide to the Intel IPP MP3 Application Programming Interface (API). As shown in Figure 10-2, the MP3 API consists of seven functions, two data structures, and several pre-defined constants and macros.

Figure 10-2 MP3 Decoder API



UnpackFrameHeader_MP3

Unpacks audio frame header.

Syntax

```
IppStatus ippUnpackFrameHeader_MP3(Ipp8u** ppBitStream, IppMP3FrameHeader*  
    pFrameHeader);
```

Parameters

<i>ppBitStream</i>	Pointer to the pointer to the first byte of the MP3 frame header (<i>*ppBitStream</i> will be updated in the function).
<i>pFrameHeader</i>	Pointer to the MP3 frame header structure.

Description

The function `ippUnpackFrameHeader_MP3` is declared in the `ippac.h` file. This function unpacks the audio frame header. If cyclic redundancy check (CRC) is enabled, this function also unpacks the CRC word.

Before calling `ippUnpackFrameHeader_MP3`, you should locate the bit stream synchronization word and ensure that **ppBitStream* points to the first byte of the 32-bit frame header.

If CRC is enabled, the assumption is that the 16-bit CRC word is adjacent to the 32-bit frame header, as defined in the MP3 standard. Before returning to the caller, the function updates the pointer **ppBitStream* so that it references the next byte after the frame header or the CRC word.

The first byte of the 16-bit CRC word is stored in `pFrameHeader->CRCWord(15:8)`, and the second byte is stored in `pFrameHeader->CRCWord(7:0)`.

The function does not detect corrupted frame headers.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>ppBitStream</i> or <i>FrameHeader</i> is NULL.

UnpackSideInfo_MP3

Unpacks side information from input bitstream for use during decoding of associated frame.

Syntax

```
IppStatus ippsUnpackSideInfo_MP3(Ipp8u** ppBitStream, IppMP3SideInfo*
    pDstSideInfo, int* pDstMainDataBegin, int* pDstPrivateBits, int* pDstScfsi,
    IppMP3FrameHeader* pFrameHeader);
```

Parameters

<i>ppBitStream</i>	Pointer to the pointer to the first byte of the side information associated with the current frame in the bit stream buffer. The function updates this parameter.
<i>pFrameHeader</i>	Pointer to the structure that contains the unpacked MP3 frame header. The header structure provides format information about the input bitstream. Both single- and dual-channel MPEG-1 and MPEG-2 modes are supported.
<i>pDstSideInfo</i>	Pointer to the MP3 side information structure. The structure contains side information that applies to all granules and all channels for the current frame. One or more of the structures are placed contiguously in the buffer pointed to by <i>pDstSideInfo</i> in the following order: {granule 0 (channel 0, channel 1), granule 1 (channel 0, channel 1)}.
<i>pDstMainDataBegin</i>	Pointer to the <code>main_data_begin</code> field.
<i>pDstPrivateBits</i>	Pointer to the <code>private bits</code> field.
<i>pDstScfsi</i>	Pointer to the scale factor selection information associated with the current frame. The data is organized contiguously in the buffer pointed to by <i>pDstScfsi</i> in the following order: {channel 0 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3), channel 1 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3)}.

Description

The function `ippsUnpackSideInfo_MP3` is declared in the `ippac.h` file. This function unpacks the side information from the input bitstream. Before calling the function `ippsUnpackSideInfo_MP3`, make sure that the pointer `*ppBitStream` points to the first byte of the bit stream that contains the side information associated with the current frame. Before returning to the caller, the function updates the pointer `*ppBitStream` so that it references the next byte after the side information.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <code>ppBitStream</code> , <code>pDstSideInfo</code> , <code>pDstMainDataBegin</code> , <code>pDstPrivateBits</code> , <code>pDstScfsi</code> , <code>pFrameHeader</code> , or <code>ppBitStream</code> is NULL.
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when some elements in the MP3 frame header structure are invalid: <code>pFrameHeader->id != IPP_MP3_ID_MPEG1</code> <code>pFrameHeader->id != IPP_MP3_ID_MPEG2</code> <code>pFrameHeader->layer != 1</code> <code>pFrameHeader->mode</code> exceeds <code>[0..3]</code> .
<code>ippStsMP3SideInfoErr</code>	Indicates an error when the value of <code>block_type</code> is zero when <code>window_switching_flag</code> is set.

UnpackScaleFactors_MP3

Unpacks scalefactors.

Syntax

```
IppStatus ippsUnpackScaleFactors_MP3_lu8s(Ipp8u** ppBitStream, int* pOffset,  
Ipp8s* pDstScaleFactor, IppMP3SideInfo* pSideInfo, int* pScfsi,  
IppMP3FrameHeader* pFrameHeader, int granule, int channel);
```

Parameters

<i>ppBitStream</i>	Pointer to the pointer to the first bitstream buffer byte associated with the scale factors for the current frame, granule, and channel. The function updates this parameter.
<i>pOffset</i>	Pointer to the next bit in the byte referenced by <i>*ppBitStream</i> . Valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit. The function updates this parameter.
<i>pSideInfo</i>	Pointer to the MP3 side information structure associated with the current granule and channel.
<i>pScfsi</i>	Pointer to scale factor selection information for the current channel.
<i>channel</i>	Channel index. Can take the values of either 0 or 1.
<i>granule</i>	Granule index. Can take the values of either 0 or 1.
<i>pFrameHeader</i>	Pointer to MP3 frame header structure for the current frame.
<i>pDstScaleFactor</i>	Pointer to the scalefactor vector for long and/or short blocks.

Description

The function `ippUnpackScaleFactors_MP3` is declared in the `ippac.h` file. This function unpacks short and/or long block scalefactors for one granule of one channel and places the results in the vector *pDstScaleFactor*. Before returning to the caller, the function updates **ppBitStream* and **pOffset* so that they point to the next available bit in the input bitstream.



NOTE. MPEG-2 intensity mode: if the intensity position is equal to the maximum value, or illegal position, the illegal position sets to negative. Consequently, in the requantization module, negative positions indicate illegal positions. Scalefactors that are not treated as intensity positions must be set to positive before using them.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>pDstScaleFactor</i> , <i>pSideInfo</i> , <i>pScfsi</i> , <i>ppBitStream</i> , or <i>pFrameHeader</i> is NULL.

<code>ippStsBadArgErr</code>	Indicates an error when <i>pOffset</i> > 7 <i>pOffset</i> < 0 granule and/or channel exceeds [0, 1].
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <i>pFrameHeader</i> → <i>id</i> != IPP_MP3_ID_MPEG1 <i>pFrameHeader</i> → <i>id</i> != IPP_MP3_ID_MPEG2 <i>pFrameHeader</i> → <i>mode</i> exceeds [0..3].
<code>ippStsMP3SideInfoErr</code>	Indicates an error when <i>pSideInfo</i> → <i>blockType</i> exceeds [0, 3] <i>pSideInfo</i> → <i>mixedBlock</i> exceeds [0, 1] <i>pSideInfo</i> → <i>sfCompress</i> exceeds [0, 15] or <i>pScfsi</i> [0..3] exceeds [0, 1] when <i>pFrameHeader</i> indicates the bitstream is MPEG-1 <i>pSideInfo</i> → <i>sfCompress</i> exceeds [0, 511] when <i>pFrameHeader</i> indicates the bitstream is MPEG-2.

HuffmanDecode_MP3

HuffmanDecodeSfb_MP3

HuffmanDecodeSfbMbp_MP3

Decodes Huffman data.

Syntax

```
IppStatus ippHuffmanDecode_MP3_lu32s(Ipp8u** ppBitStream, int* pOffset,  
    Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo* pSideInfo,  
    IppMP3FrameHeader* pFrameHeader, int hufSize);  
  
IppStatus ippHuffmanDecodeSfb_MP3_lu32s(Ipp8u** ppBitStream, int* pOffset,  
    Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo* pSideInfo,  
    IppMP3FrameHeader* pFrameHeader, int hufSize,  
    IppMP3ScaleFactorBandTableLong* pSfbTableLong);
```

```
IppStatus, ippsHuffmanDecodeSfbMbp_MP3_lu32s(Ipp8u** ppBitStream, int* pOffset,
Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo* pSideInfo,
IppMP3FrameHeader* pFrameHeader, int hufSize,
IppMP3ScaleFactorBandTableLong* pSfbTableLong,
IppMP3ScaleFactorBandTableShort pSfbTableShort,
IppMP3MixedBlockPartitionTable pMbpTable);
```

Parameters

<i>ppBitStream</i>	Pointer to the pointer to the first bit stream byte that contains the Huffman code words associated with the current granule and channel. The function updates this parameter.
<i>pOffset</i>	Pointer to the starting bit position in the bit stream byte pointed to by <i>*ppBitStream</i> . The parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit, and 7 corresponds to the least significant bit. The function updates this parameter.
<i>pSideInfo</i>	Pointer to MP3 structure containing side information associated with the current granule and channel.
<i>pFrameHeader</i>	Pointer to MP3 structure containing the header associated with the current frame.
<i>pDstIs</i>	Pointer to the vector of decoded Huffman symbols used to compute the quantized values of the 576 spectral coefficients associated with the current granule and channel.
<i>pDstNonZeroBound</i>	Pointer to the spectral region above which all coefficients are set to zero.
<i>hufSize</i>	The number of Huffman code bits associated with the current granule and channel.
<i>pSfbTableLong</i>	Pointer to the scale factor bands table for a long block.
<i>pSfbTableShort</i>	Pointer to scale factor band table for short block. You can use the default table from MPEG-1, MPEG-2 standards.
<i>pMbpTable</i>	Pointer to the mixed block partition table.

Description

The functions `ippsHuffmanDecode_MP3`, `ippsHuffmanDecodeSfb_MP3`, and `ippsHuffmanDecodeSfbMbp_MP3_lu32s` are declared in the `ippac.h` file. These functions decode Huffman symbols for the 576 spectral coefficients associated with one granule of one channel. Before returning to the caller, the function updates **ppBitStream* so that it points to the byte in the bitstream that contains the first new bit following the decoded block of Huffman codes.

The function `ippsHuffmanDecodeSfb_MP3` makes the use of a predefined scale factor bands table possible when working with a long block. Alternatively, you can use the default table from MPEG-1, or MPEG-2 standards.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <code>ppBitStream</code> , <code>pOffset</code> , <code>pDstIs</code> , <code>pDstNonZeroBound</code> , <code>pSideInfo</code> , <code>pFrameHeader</code> , or <code>ppBitStream</code> is NULL or when <code>pOffset < 0</code> or <code>pOffset > 7</code> .
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when some elements in the MP3 frame header structure are invalid.
<code>ippStsMP3SideInfoErr</code>	Indicates an error when some elements in the MP3 side information structure are invalid or when <code>hufSize < 0</code> or <code>hufSize > pSideInfo->part23Len</code> .
<code>ippStsErr</code>	Indicates unknown error.

ReQuantize_MP3 ReQuantizeSfb_MP3

Requantizes the decoded Huffman symbols.

Syntax

```
IppStatus ippsReQuantize_MP3_32s_I(Ipp32s* pSrcDstIsXr, int* pNonZeroBound,
    Ipp8s* pScaleFactor, IppMP3SideInfo* pSideInfo, IppMP3FrameHeader*
    pFrameHeader, Ipp32s* pBuffer);

IppStatus ippsReQuantizeSfb_MP3_32s_I(Ipp32s* pSrcDstIsXr, int* pNonZeroBound,
    Ipp8s* pScaleFactor, IppMP3SideInfo* pSideInfo, IppMP3FrameHeader*
    pFrameHeader, Ipp32s* pBuffer, IppMP3ScaleFactorBandTableLong pSfbTableLong,
    IppMP3ScaleFactorBandTableShort pSfbTableShort);
```

Parameters

<code>pSrcDstIsXr</code>	Pointer to the vector of the decoded Huffman symbols. For stereo and dual channel modes, right channel data begins at the address <code>&(pSrcDstIsXr[576])</code> . The function updates this vector.
--------------------------	--

<i>pNonZeroBound</i>	Pointer to the spectral bound above which all coefficients are set to zero. For stereo and dual channel modes, the left channel bound is <i>pNonZeroBound</i> [0], and the right channel bound is <i>pNonZeroBound</i> [1].
<i>pScaleFactor</i>	Pointer to the scalefactor buffer. For stereo and dual channel modes, the right channel scalefactors begin at & (<i>pScaleFactor</i> [IPP_MP3_SF_BUF_LEN]).
<i>pSideInfo</i>	Pointer to the side information for the current granule.
<i>pFrameHeader</i>	Pointer to the frame header for the current frame.
<i>pBuffer</i>	Pointer to the workspace buffer. The buffer length must be 576 samples.
<i>pSfbTableLong</i>	Pointer to the scale factor bands table for a long block.
<i>pSfbTableShort</i>	Pointer to the scale factor bands table for a short block.

Description

The functions `ippsReQuantize_MP3` and `ippsReQuantizeSfb_MP3` are declared in the `ippac.h` file. These functions requantize the decoded Huffman symbols. Spectral samples for the synthesis filter bank are derived from the decoded symbols using the requantization equations given in the ISO standard.

If necessary, you can apply stereophonic mid/side (M/S) and/or intensity decoding. The function returns requantized spectral samples in the vector *pSrcDstIsXr*.

For short blocks you can perform reordering operation. You should preallocate a workspace buffer pointed to by *pBuffer* prior to calling the requantization function. For short blocks, the value pointed to by *pNonZeroBound* is recalculated according to the reordered sequence.

You can use predefined scale factor bands table for a long or short block with the function `ippsReQuantizeSfb_MP3`. Alternatively, you can use the default table from MPEG-1, or MPEG-2 standards.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>pSrcDstIsXr</i> , <i>pNonZeroBound</i> , <i>pScaleFactor</i> , <i>pSideInfo</i> , <i>pFrameHeader</i> , or <i>pBuffer</i> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>pNonZeroBound</i> exceeds [0 , 576].

<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <i>pFrameHeader->id</i> != IPP_MP3_ID_MPEG1 <i>pFrameHeader->id</i> != IPP_MP3_ID_MPEG2 <i>pFrameHeader->samplingFreq</i> exceeds [0, 2], <i>pFrameHeader->mode</i> exceeds [0, 3], <i>pFrameHeader->modeExt</i> exceeds [0, 3].
<code>ippStsMP3SideInfoErr</code>	Indicates an error when the bitstream is in the stereo mode, but the block type of left is different from that of right, <i>pSideInfo[ch].blockType</i> exceeds [0, 3], <i>pSideInfo[ch].mixedBlock</i> exceeds [0, 1], <i>pSideInfo[ch].globGain</i> exceeds [0, 255], <i>pSideInfo[ch].sfScale</i> exceeds [0, 1], <i>pSideInfo[ch].preFlag</i> exceeds [0, 1], <i>pSideInfo[ch].pSubBlkGain[w]</i> exceeds [0, 7], where <i>ch</i> is within the range of 0 to 1, and <i>w</i> is within the range of 0 to 2.
<code>ippStsErr</code>	Indicates an unknown error.

MDCTInv_MP3

*Performs the first stage of
hybrid synthesis filter bank.*

Syntax

```
IppStatus ippMDCTInv_MP3_32s(Ipp32s* pSrcXr, Ipp32s* pDstY, Ipp32s*  
    pSrcDstOverlapAdd, int nonZeroBound, int* pPrevNumOfImdct, int blockType,  
    int mixedBlock);
```

Parameters

<i>pSrcXr</i>	Pointer to the vector of requantized spectral samples for the current channel and granule, represented in Q5.26 format.
<i>pDstY</i>	Pointer to the vector of IMDCT outputs in Q7.24 format for input to PQMF bank.

<i>pSrcDstOverlapAdd</i>	Pointer to the overlap-add buffer. Contains the overlapped portion of the previous granule IMDCT output in Q7.24 format. The function updates this buffer.
<i>nonZeroBound</i>	Limiting bound for spectral coefficients. All spectral coefficients exceeding this boundary become zero for the current granule and channel.
<i>pPrevNumOfImdct</i>	Pointer to the number of IMDCTs computed for the current channel of the previous granule. The function updates this parameter so that it references the number of IMDCTs for the current granule.
<i>blockType</i>	Block type indicator.
<i>mixedBlock</i>	Mixed block indicator.

Description

The function `ippsMDCTInv_MP3` is declared in the `ippac.h` file. This function performs the first stage of the hybrid synthesis filter bank. The following operations are performed:

- alias reduction
- inverse modified discrete cosine transform (IMDCT) according to block size specifiers and mixed block modes
- overlap add of IMDCT outputs
- frequency inversion prior to pseudo-quadrature mirror synthesis filter (PQMF) bank.

Since IMDCT is a lapped transform, you should preallocate a buffer referenced by *pSrcDstOverlapAdd* to maintain the IMDCT overlap-add state.

The buffer must contain 576 elements. Prior to the first call to the synthesis filter bank, all elements of the overlap-add buffer should be set to zero. In between all subsequent calls, the MP3 application must preserve the contents of the overlap-add buffer.

Upon entry to `ippsMDCTInv_MP3_32s`, the overlap-add buffer should contain the IMDCT output generated by operating on the previous granule. Upon exit from `ippsMDCTInv_MP3_32s`, the overlap-add buffer contains the overlapped portion of the output generated by operating on the current granule.

Upon return from the function, the IMDCT sub-band output samples are organized as follows:

$pDstY[j*32+subband]$, for $j = 0$ to 17, $subband = 0$ to 31.

Q5.26 designates that 5 bits before and 26 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q7.24 designates that 7 bits before and 24 bits after fixed point position are used to present a 32-bit value in the fixed point format.



NOTE. Note that the pointers *pSrcXr* and *pDstY* must reference different buffers.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsErr</code>	Indicates an error when one or more of the following input data errors are detected: either <i>blockType</i> exceeds <code>[0, 3]</code> , or <i>mixedBlock</i> exceeds <code>[0, 1]</code> , or <i>nonZeroBound</i> exceeds <code>[0, 576]</code> , or <i>*pPrevNumOfImdct</i> exceeds <code>[0, 32]</code> .

SynthPQMF_MP3

Performs the second stage of hybrid synthesis filter bank.

Syntax

```
IppStatus ippsSynthPQMF_MP3_32s16s(Ipp32s* pSrcY, Ipp16s* pDstAudioOut, Ipp32s*
    pVBuffer, int* pVPosition, int mode);
```

Parameters

<i>pSrcY</i>	Pointer to the block of 32 IMDCT sub-band input samples in Q7.24 format.
--------------	--

<i>pDstAudioOut</i>	Pointer to the block of 32 reconstructed PCM output samples in 16-bit signed little-endian format. Left and right channels are interleaved according to the <i>mode</i> flag.
<i>pVBuffer</i>	Pointer to the input workspace buffer containing Q7.24 data. The function updates this parameter.
<i>pVPosition</i>	Pointer to the internal workspace index. The function updates this parameter.
<i>mode</i>	Flag that indicates whether or not PCM audio output channels should be interleaved. 1 indicates not interleaved, 2 indicates interleaved.

Description

The function `ippsSynthPQMF_MP3` is declared in the `ippac.h` file. This function performs the second stage of the hybrid synthesis filter bank, that is, a critically-sampled 32-channel PQMF synthesis bank that generates 32 time-domain output samples for each 32-sample input block of IMDCT outputs.

For each input block, the PQMF generates an output sequence of 16-bit signed little-endian PCM samples in the vector pointed to by *pDstAudioOut*.

If *mode* equals to 2, the left and right channel output samples are interleaved, that is, LRLRLR, so that the left channel data is organized as follows:

pDstAudioOut [2*i], i = 0 to 31.

If *mode* equals 1, then the left and right channel outputs are not interleaved.

Since PQMF bank contains memory, the MP3 application must maintain two state variables in between calls to the function.

The application must preallocate a workspace buffer of size 512x (Number of Channels) for the PQMF computation. This buffer is referenced by the pointer *pVBuffer* and its elements should be initialized to zero prior to the first call. During subsequent calls, the *pVBuffer* input for the current call should contain the *pVbuffer* output generated by the previous call.

In addition to *pVBuffer*, the MP3 application must also initialize to zero and thereafter preserve the value of the state variable *pVPosition*. The MP3 application should modify the values contained in *pVBuffer* or *pVPosition* only during decoder reset, and the reset values should always be zero.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippNullPtrErr</code>	Indicates an error when least one of the pointers <i>pSrcY</i> , <i>pDstAudioOut</i> , <i>pVBuffer</i> , or <i>pVPosition</i> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when at least one of the specified pointers is NULL, or the value of <i>mode</i> is less than 1 or more than 2, or when the value of <i>pVPosition</i> exceeds [0, 15].
<code>ippStsErr</code>	Indicates an unknown error.

Advanced Audio Coding Functions

The ISO/IEC 13818-7 MPEG-2 AAC (Advanced Audio Coding) algorithm is an efficient coding method for surround signals like 5-channel signals (left, right, center, left surround, right surround).

MPEG-2 AAC supports up to 48 main audio channels with the sampling frequency between 8kHz and 96kHz. MPEG formal tests have shown that for 5-channel audio signals, AAC satisfies the ITU-R quality requirements and provides slightly better audio quality at 320 kilobits per second (kbps) than MPEG-2 BC (Backwards Compatibility) provides at 640 kbps.

Due to its high coding efficiency, AAC is a prime candidate for any digital broadcasting system and has been selected by the DRM (Digital Radio Mondiale) system. AAC also plays a major role for the delivery of high quality music via the Internet. Moreover, with some modifications, AAC is the only high quality audio coding scheme adopted within the MPEG-4 standard, the future “global multimedia language”.

Global Macros

[Table 10-4](#) shows the definitions of global macros.

Table 10-4 Global Macro Definitions

Global Macro Name	Definition	Notes
IPP_AAC_FRAME_LEN	1024	size of data in one frame
IPP_AAC_SF_LEN	120	scale factor buffer length
IPP_AAC_GROUP_NUM_MAX	8	maximum group number for one frame
IPP_AAC_TNS_COEF_LEN	60	TNS coefficients buffer length
IPP_AAC_TNS_FILT_MAX	8	maximum TNS filter number for one frame
IPP_AAC_PRED_SFB_MAX	41	maximum prediction scale factor bands number for one frame
IPP_AAC_ELT_NUM	16	maximum number of elements for one program
IPP_AAC_LFE_ELT_NUM	4	maximum Low Frequency Enhance elements number for one program

Table 10-4 **Global Macro Definitions** (continued)

Global Macro Name	Definition	Notes
IPP_AAC_DATA_ELT_NUM	8	maximum data elements number for one program
IPP_AAC_COMMENTS_LEN	256	maximum length of the comment field in bytes
IPP_AAC_PULSE_NUM_MAX	5	maximum number of the pulse escapes

Data Types and Structures

This section describes the data types and structures of the Intel® IPP advanced audio coding.

ADIF Header

```
typedef struct {
    Ipp32u ADIFId;      /* 32-bit, "ADIF" ASCII code */
    int copyIdPres;     /* copy id flag: 0 - off, 1 - on */
    int originalCopy;   /* copyright bit:
                        0 - no copyright on the coded bitstream,
                        1 - copyright protected */
    int home;           /* original/home bit:
                        0 - the bitstream is a copy,
                        1 - the bitstream is the original
    int bitstreamType; /* bitstream flag:
                        0 - constant rate bitstream,
                        1 - variable rate bitstream */
    int bitRate;        /* bit rate: if 0, unknown bit rate */
    int numPrgCfgElt;   /* number of program configure elements */
    int pADIFBufFullness[IPP_AAC_ELT_NUM]; /* buffer fullness */
    Ipp8u pCopyId[9];   /* 72-bit copy id */
} IppAACADIFHeader;
```


ADTS Frame Header

```
typedef struct {
    /* ADTS fixed header */
    int id;                /* ID 1*/
    int layer;              /* layer index: 0x3 - Layer I
                           0x2 - Layer II
                           0x1 - Layer III */
    int protectionBit;      /* CRC flag 0: CRC on, 1: CRC off */
    int profile;             /* profile: 0:MP, 1:LC, 2:SSR */
    int samplingRateIndex;  /* sampling rate index */
    int privateBit;         /* private_bit, no use */
    int chConfig;           /* channel configure */
    int originalCopy;       /* copyright bit:
                           0 - no copyright on the coded bitstream,
                           1 - copyright protected */
    int home;               /* original/home bit:
                           0 - the bitstream is a copy,
                           1 - the bitstream is the original
    int emphasis;           /* not used by ISO/IEC 13818-7, but used
                           by 14496-3 */

    /* ADTS variable header */
    int cpRightIdBit;       /* copyright id bit */
    int cpRightIdStart;     /* copyright id start */
    int frameLen;           /* frame length in bytes */
    int ADTSBufFullness;    /* buffer fullness */
    int numRawBlock;        /* number of raw data blocks in the
                           frame */
    /* ADTS CRC error check, 16bits */
    int CRCWord;            /* CRC-check word */
} IppAACADTSFrameHeader;
```

Individual Channel Side Information

```
typedef struct {
    /* unpacked from the bitstream */
    int icsReservedBit;      /* reserved bit */
    int winSequence;         /* window sequence flag */
    int winShape;            /* window shape flag:
                               0 - sine window,
                               1 - KBD window */
    int maxSfb;              /* maximum effective scale factor bands */
    int sfGrouping;          /* scale factor grouping information */
    int predDataPres;        /* prediction data present flag for one
                               frame:
                               0 - prediction off,
                               1 - prediction on */
    int predReset;           /* prediction reset flag:
                               0 - reset off,
                               1 - reset on */
    int predResetGroupNum;   /* prediction reset group number */
    Ipp8u pPredUsed[IPP_AAC_PRED_SFB_MAX+3]; /* prediction flag buffer
                                               for each scale factor band:
                                               0 - off,
                                               1 - on buffer length 44 bytes, 4-byte
                                               align */
    /* decoded from the above info */
    int numWinGrp;           /* group number */
    int pWinGrpLen[IPP_AAC_GROUP_NUM_MAX]; /* buffer for number of
                                               windows in each group */
} IppAACIcsInfo;
```

AAC Scalable Main Element Header

```
typedef struct{
    int windowSequence; /* windows is short or long type */
    int windowShape;    /* what window is used for the right hand
```

```

                                part of this analysis window */
int maxSfb;                    /* number of scale factor band transmitted */
int sfGrouping;               /* grouping of short spectral data */
int numWinGrp;                /* window group number */
int pWinGrpLen[IPP_AAC_GROUP_NUM_MAX]; /* length of every group */
int msMode;                   /* MS stereo flag:
                                0 - none,
                                1 - different for every sfb,
                                2 - all */
Ipp8u (*ppMsMask)[IPP_AAC_SF_MAX]; /* if MS's used in one sfb, when
                                     msMode ==1 */
IppAACTnsInfo pTnsInfo[IPP_AAC_CHAN_NUM]; /* TNS structure for
                                             channels */
IppAACLtpInfo pLtpInfo[IPP_AAC_CHAN_NUM]; /* LTP structure for
                                             channels */
}IppAACMainHeader;

```

AAC Scalable Extension Element Header

```

typedef struct{
    int msMode;    /* 0 - non; 1 - part; 2 - all */
    int maxSfb;    /* number of scale factor band for extension layer */
    Ipp8u (*ppMsMask)[IPP_AAC_SF_MAX]; /* if ms is used */
    IppAACTnsInfo pTnsInfo[IPP_AAC_CHAN_NUM]; /* TNS structure for
                                             Stereo */
    int pDiffControlLr[IPP_AAC_CHAN_NUM][IPP_AAC_SFB_MAX]; /* FSS
                                                             information for stereo */
}IppAACExtHeader;

```

AAC Program Config Element

```

typedef struct {
    int eltInstTag;           /* element instanse tag */
    int profile;              /* profile index:
                                0 - main,
                                1 - Low Complexity,

```

```

                                2 - Scaleable Sampling Rate,
                                3 - reserved */

int  samplingRateIndex;        /* sampling rate index */
int  numFrontElt;              /* number of front elements */
int  numSideElt;               /* number of side elements */
int  numBackElt;               /* number of back elements */
int  numLfeElt;                /* number of LFE elements */
int  numDataElt;               /* number of data elements */
int  numValidCcElt;            /* number of coupling channel elements */
int  monoMixdownPres;          /* mono mixdown flag:
                                0 - off,
                                1 - on */

int  monoMixdownEltNum;        /* number of mono mixdown elements */
int  stereoMixdownPres;        /* stereo mixdown flag:
                                0 - off,
                                1 - on */

int  stereoMixdownEltNum;      /* number of stereo mixdown elements */
int  matrixMixdownIdxPres;     /* matrix mixdown flag:
                                0 - off,
                                1 - on */

int  matrixMixdownIdx;         /* identifier of the surround mixdown
                                coefficient */

int  pseudoSurroundEnable;     /* pseudo surround:
                                0 - off,
                                1 - on */

int  pFrontIsCpe[IPP_AAC_ELT_NUM]; /* channel pair flag for
                                front elements */
int  pFrontTagSel[IPP_AAC_ELT_NUM]; /* instance tag for front
                                elements */
int  pSideIsCpe[IPP_AAC_ELT_NUM]; /* channel pair flag for
                                side elements */
int  pSideTagSel[IPP_AAC_ELT_NUM]; /* instance tag for side
                                elements */

```

```

int pBackIsCpe[IPP_AAC_ELT_NUM];      /* channel pair flag for
                                         back elements */
int pBackTagSel[IPP_AAC_ELT_NUM];      /* instance tag for back
                                         elements */
int pLfeTagSel[IPP_AAC_LFE_ELT_NUM];   /* channel pair flag for
                                         LFE elements */
int pDataTagSel[IPP_AAC_DATA_ELT_NUM]; /* instance tag for data
                                         elements */
int pCceIsIndSw[IPP_AAC_ELT_NUM];      /* independent flag for
                                         coupling */

/* channel elements */
int pCceTagSel[IPP_AAC_ELT_NUM];      /* instance tag for coupling
                                         channel elements */
int numComBytes;                       /* number of comment field
                                         bytes */

Ipp8s pComFieldData[IPP_AAC_COMMENTS_LEN]; /* comment buffer field */
}IppAACPrGcFgElt;

```

Temporal Noise Shaping (TNS) Structure for One Layer

```

typedef struct{
    int tnsDataPresent; /* Indicator of TNS data present:
                        0 - not present,
                        1 - present */
    int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX]; /*TNS number filter buffer*/
    int pTnsFiltCoefRes[IPP_AAC_GROUP_NUM_MAX]; /* TNS coef resolution
                                                flag */
    int pTnsRegionLen[IPP_AAC_TNS_FILT_MAX]; /* TNS filter length */
    int pTnsFiltOrder[IPP_AAC_TNS_FILT_MAX]; /* TNS filter order */
    int pTnsDirection[IPP_AAC_TNS_FILT_MAX]; /* TNS filter direction
                                                flag */
    int pTnsCoefCompress[IPP_AAC_GROUP_NUM_MAX]; /* The most
                                                significant bit of the coefficients of the noise
                                                shaping filter in window w is omitted or not */
}

```

```

    Ipp8s pTnsFiltCoef[IPP_AAC_TNS_COEF_LEN]; /* Coefficients of one
                                              noise shaping filter applied to window w */
}IppAACTnsInfo;

```

Long Term Prediction (LTP) Structure

```

typedef struct{
    int ltpDataPresent;      /* Indicator of LTP data present:
                               0 - not present,
                               1 - present */
    int ltpLag;              /* the optimal delay from 0 to 2047 */
    Ipp16s ltpCoef;          /* indicate the LTP coefficient */
    int pLtpLongUsed[IPP_AAC_SF_MAX+1]; /* if long block use ltp */
    int pLtpShortUsed[IPP_AAC_WIN_MAX]; /* if short block use ltp */
    int pLtpShortLagPresent[IPP_AAC_WIN_MAX]; /* if short lag is
                                              transmitted */
    int pLtpShortLag[IPP_AAC_WIN_MAX]; /* relative delay for short
                                              window */
}IppAACLtpInfo;

```

Channel Pair Element

```

typedef struct {
    int commonWin;           /* common window flag:
                               0 - off,
                               1 - on */
    int msMaskPres;          /* MS stereo mask present flag */
    Ipp8u pMsMask[IPP_AAC_GROUP_NUM_MAX][IPP_AAC_SF_MAX]; /* MS stereo
                                                              flag buffer for each scalefactor
                                                              band */
} IppAACChanPairElt;

```

Channel Information

```
typedef struct {  
    int tag;                /* channel tag */  
    int id;                 /* element id */  
    int samplingRateIndex;  /* sampling rate index */  
    int predSfbMax;         /* maximum prediction scale factor bands*/  
    int preWinShape;        /* previous block window shape */  
    int winLen;             /* 128 - short window, 1024 - others */  
    int numWin;             /* 1 - long block, 8 - short block */  
    int numSwb;             /* decided by sampling frequency and block  
                           type */  
  
    /* unpacking from the bitstream */  
    int globGain;           /* global gain */  
    int pulseDataPres;     /* pulse data present flag: 0 - off, 1 - on */  
    int tnsDataPres;       /* TNS data present flag: 0 - off, 1 - on */  
    int gainContrDataPres; /* gain control data present flag: 0 - off,  
                           1 - on */  
  
    /* icsInfo pointer */  
    IppAACIcsInfo *pIcsInfo; /* pointer to IppAACIcsInfo structure */  
  
    /* channel pair pointer */  
    IppAACChanPairElt *pChanPairElt; /* pointer to IppAACChanPairElt  
                                       structure */  
  
    /* section data */  
    Ipp8u pSectCb[IPP_AAC_SF_LEN]; /* section code book buffer */  
    Ipp8u pSectEnd[IPP_AAC_SF_LEN]; /* end of scalefactor offset  
                                       in each section */  
    int pMaxSect[IPP_AAC_GROUP_NUM_MAX]; /* maximum section number  
                                       for each group */  
    int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX]; /*TNS number filter  
                                       number buffer*/  
}
```

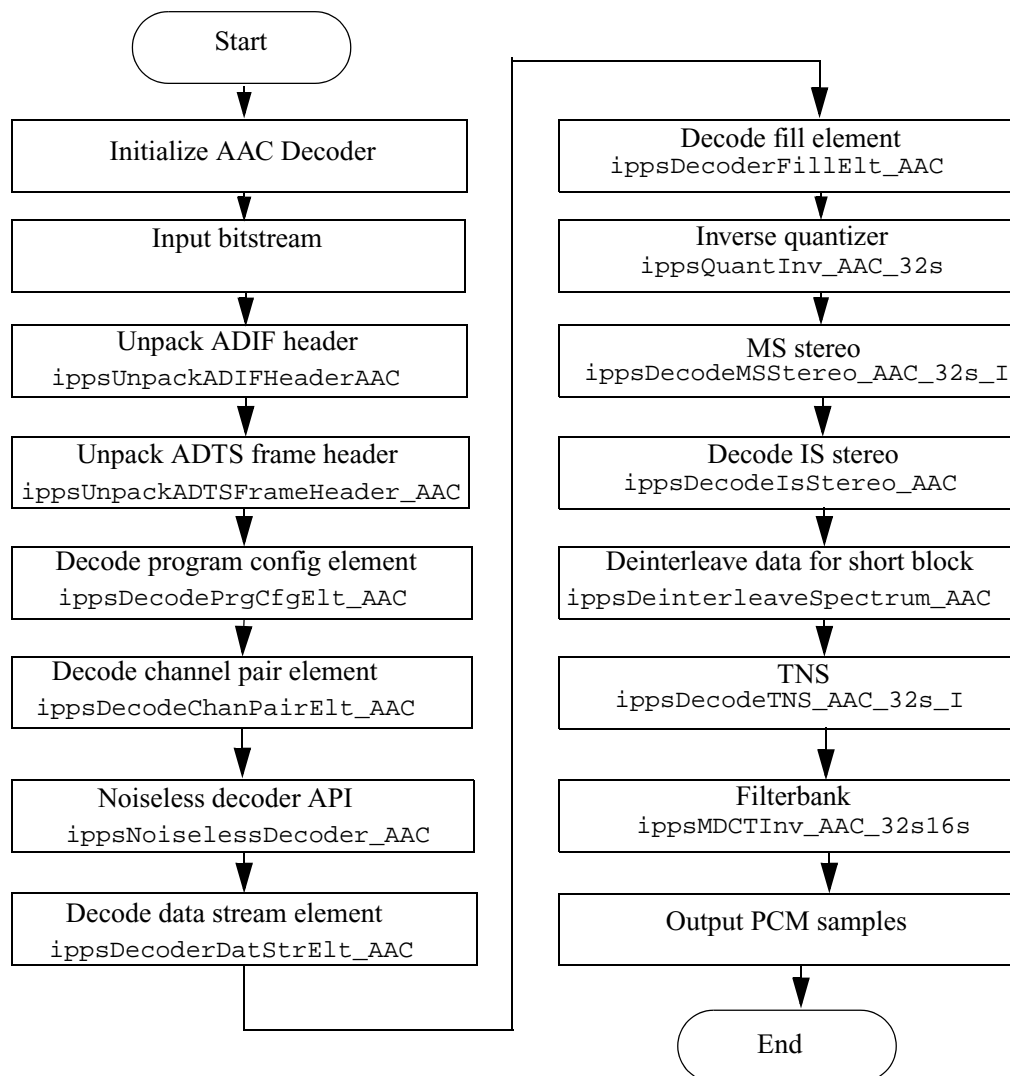
```
/* TNS data */
int pTnsFiltCoefRes[IPP_AAC_GROUP_NUM_MAX];/* TNS coefficients
resolution flag */

int pTnsRegionLen[IPP_AAC_TNS_FILT_MAX];/* TNS filter length */
int pTnsFiltOrder[IPP_AAC_TNS_FILT_MAX];/* TNS filter order */
int pTnsDirection[IPP_AAC_TNS_FILT_MAX];/* TNS filter direction
flag */
}IppAACChanInfo;
```


AAC Decoding Primitives

The AAC decoder API provides a variety of AAC LC decoder functions, including bitstream unpacking and AAC core decoding functions. This provides customers great flexibility in configuring the decoder system. See [ISO/IEC 13818-7:1997](https://www.iso.org/standard/54461.html).

Figure 10-3 AAC Decoding Flowchart



MPEG-2 AAC Primitives

In the following sections, the parameter *maxSfb* designates the number of scale factor bands transmitted per group and unpacked from the bitstream. The parameter *numSwb* stands for the number of scale factor window bands for the short block or the number of scale factor window bands for the long block. The values are computed according to the sampling rate and the block type.

See clause 8.3.1 of [ISO/IEC 13818-7:1997](#).

UnpackADIFHeader_AAC

Gets the AAC ADIF format header.

Syntax

```
IppStatus ippUnpackADIFHeader_AAC(Ipp8u** ppBitStream, IppAACADIFHeader*  
    pADIFHeader, IppAACPrnCfElt* pPrnCfElt, int prnCfEltMax);
```

Parameters

<i>ppBitStream</i>	Double pointer to the current byte before the ADIF header.
<i>pADIFHeader</i>	Pointer to the <code>IppAACADIFHeader</code> structure.
<i>pPrnCfElt</i>	Pointer to the <code>IppAACPrnCfElt</code> structure. There must be <code>prnCfEltMax</code> elements in the buffer.
<i>prnCfEltMax</i>	Maximum program configure element number. Must be within the range of [1, 16].

Description

This function is declared in the `ippac.h` file. The function gets the AAC ADIF format header, including program configuration elements from the input bitstream. See Table 6.2, and 6.21. of [ISO/IEC 13818-7:1997](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pADIFHeader</i> , <i>pPrnCfElt</i> , or <i>*ppBitStream</i> is NULL.

`ippStsAacPrgNumErr` Indicates an error when the decoded `pADIFHeader->numPrgCfgElt` > `prgCfgEltMax`, or `prgCfgEltMax` is outside the range of `[1, IPP_AAC_MAX_ELT_NUM]`.



NOTE. `pADIFHeader->numPrgCfgElt` is the number directly unpacked from bitstream plus 1.

UnpackADTSFrameHeader_AAC

Gets ADTS frame header from the input bitstream.

Syntax

```
IppStatus ippUnpackADTSFrameHeader_AAC(Ipp8u** ppBitStream,
    IppAACADTSFrameHeader* pADTSFrameHeader);
```

Parameters

<code>ppBitStream</code>	Double pointer to the current byte after unpacking the ADTS frame header.
<code>pADTSFrameHeader</code>	Pointer to the <code>IppAACADTSFrameHeader</code> structure.

Description

The function gets ADTS frame header from the input bitstream. If the CRC word is applied, the first byte of the 16-bit CRC word is stored in `pADTSFrameHeader->CRCWord[15:8]` and the second byte is stored in `pADTSFrameHeader->CRCWord[7:0]`. It does not check whether the header is corrupt.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <code>ppBitStream</code> , <code>pADTSFrameHeader</code> or <code>*ppBitStream</code> is NULL.

DecodePrgCfgElt_AAC

Gets program configuration element from the input bitstream.

Syntax

```
IppStatus ippDecodePrgCfgElt_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACPrGCfgElt* pPrgCfgElt);
```

Parameters

<i>ppBitStream</i>	Double pointer to the current byte after decoding the program configuration element.
<i>pOffset</i>	Pointer to the bit position in the byte pointed to by <i>*ppBitStream</i> . Valid within 0 to 7: 0 stands for the most significant bit of the byte; 7 stands for the least significant bit of the byte.
<i>pPrgCfgElt</i>	Pointer to IppAACPrGCfgElt structure.

Description

This function is declared in the `ippac.h` file. The function gets the program configuration element from the input bitstream. See clause 8.5 and Table 6.21 of [ISO/IEC 13818-7](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>pPrgCfgElt</i> , or <i>*ppBitStream</i> is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of the range of [0, 7].

DecodeChanPairElt_AAC

Gets channel_pair_element from the input bitstream.

Syntax

```
IppStatus ippDecodeChanPairElt_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACIcsInfo* pIcsInfo, IppAACChanPairElt* pChanPairElt, int predSfbMax);
```

Parameters

<i>ppBitStream</i>	Double pointer to the current byte after decoding the channel pair element.
<i>pOffset</i>	Pointer to the bit position in the byte pointed to by <i>ppBitStream</i> . Valid within 0 to 7: 0 stands for the most significant bit of the byte; 7 stands for the least significant bit of the byte.
<i>pIcsInfo</i>	Pointer to <i>IppAACIcsInfo</i> structure. If <i>pIcsInfo->predDataPres</i> = 0, set <i>pIcsInfo->predReset</i> = 0. Only the first <i>pIcsInfo->numWinGrp</i> elements in <i>pIcsInfo->pWinGrpLen</i> are meaningful. You should not change some members of the structure, as shown in Table 10-5 .
<i>pChanPairElt</i>	Pointer to <i>IppAACChanPairElt</i> structure. You should not change some members of the structure, as shown in Table 10-5 .
<i>predSfbMax</i>	Maximum prediction scale factor bands. For LC profile, set <i>predSfbMax</i> = 0.

Description

This function is declared in the *ippac.h* file. The function gets the channel pair element from the input bitstream. Individual channel stream is not included.

If *common_window* flag decoded from the input bitstream is 0, all members of *pIcsInfo* and *pChanPairElt* remain unchanged except for *pChanPairElt->commonWin*.

See clause 8.3 and Table 6.10, 6.11 of [ISO/IEC 13818-7:1997](#).

Table 10-5 **Unchanged Members of *plcsInfo***

Unchanged Members	Conditions
<code>sfGrouping</code>	<code>pIcsInfo->winSequence != 2</code>
<code>predResetGroupNum</code>	<code>pIcsInfo->predDataPres == 0 </code> <code>pIcsInfo->predReset == 0</code>
<code>pPredUsed[sfb]</code>	<code>pIcsInfo->predDataPres == 0</code>
<code>pMsUsed[sfb]</code>	<code>pChanPairElt->msMaskPres != 1</code>

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>*ppBitStream</i> , <i>pIcsInfo</i> , or <i>pChanPairElt</i> is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of the range of [0, 7].
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>pIcsInfo->maxSfb</i> decoded from bitstream is greater than IPP_AAC_MAX_SFB, the maximum scale factor band for all sampling frequencies.
<code>ippStsAacPredSfbErr</code>	Indicates an error when <i>predSfbMax</i> is out of the range of [0, IPP_AAC_PRED_SFB_MAX].

NoiselessDecoder_LC_AAC

Decodes all data for one channel.

Syntax

```
IppStatus ippsNoiselessDecoder_LC_AAC(Ipp8u** ppBitStream, int* pOffset, int
commonWin, IppAACChanInfo* pChanInfo, Ipp16s* pDstScalefactor, Ipp32s*
pDstQuantizedSpectralCoef, Ipp8u* pDstSfbCb, Ipp8s* pDstTnsFiltCoef);
```

Parameters

<i>ppBitStream</i>	Double pointer to the current byte.
<i>pOffset</i>	Pointer to the offset in one byte.
<i>commonWin</i>	Common window indicator.
<i>pChanInfo</i>	Pointer to the channel information. <code>IppAACChanInfo</code> structure. Denotes <i>pIcsInfo</i> as <i>pChanInfo->pIcsInfo</i> as shown in Table 10-6 .
<i>pDstScalefactor</i>	Pointer to the scalefactor or intensity position buffer. Buffer length is more than or equal to 120. Only <i>maxSfb</i> elements are stored for each group. There is no space between sequence groups.
<i>pDstQuantizedSpectralCoef</i>	Pointer to the quantized spectral coefficients data. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length is more than or equal to 1024.
<i>pDstSfbCb</i>	Pointer to the scale factor band codebook. Buffer length must be more than or equal to 120. Store <i>maxSfb</i> elements for each group. There is no space between the sequence groups.
<i>pDstTnsFiltCoef</i>	Pointer to TNS coefficients. Buffer length must be more than or equal to 60. The store sequence is TNS order elements for each filter for each window. The elements are not changed if the corresponding TNS order is zero.

Description

This function is declared in the `ippac.h` file. The function decodes all data for one channel, including scale factor/intensity positions, spectral coefficients, TNS coefficients, and associated side information for LC profile.

You need to set *pChanInfo->pIcsInfo*, *pChanInfo->samplingRateIndex*, *pChanInfo->predSfbMax* to correct pointer/values before calling this function.

Table 10-6 Input/Output Members List of *pChanInfo*

Member	Output
<i>Tag</i>	Not used.
<i>id</i>	Not used.
<i>preWinShape</i>	Not used.
<i>pChanPairElt</i>	Not used.
<i>samplingRateIndex</i>	As input. Not changed.
<i>predSfbMax</i>	As input. Must be 0. Not changed.
<i>winLen</i>	As output. Set to 128, if decoded, <i>pIcsInfo->winSequence</i> is short block. Otherwise set to 1024.
<i>numWin</i>	As output. Set to 8, if decoded <i>plcsInfo->winSequence</i> is short block. Otherwise set to 1.
<i>numSwb</i>	As output. Set to the maximum number of scale factor window bands in each group according to <i>samplingRateIndex</i> and <i>pIcsInfo->winSequence</i> . See Table 8.4-8.1 of ISO/IEC 13818-7:1997 .
<i>globGain</i> <i>pulseDataPres</i> <i>tnsDataPres</i> <i>gainContrDataPres</i>	As output. Unpacked from bitstream.
<i>pMaxSect</i>	As output. Pointer to the maximum of sections number in each group. Only <i>pIcsInfo->numWinGrp</i> elements in the buffer are meaningful.
<i>pSectCb</i>	As output. Pointer to the section codebook. Only <i>pMaxSect[g]</i> elements are stored for each group. There is no space between the sequence groups.
<i>pTnsRegionLen</i>	As output. Pointer to the length of the region in units of scale factor bands to which one filter is applied in each window.
<i>pTnsFiltOrder</i>	As output. Pointer to the order of the temporal noise shaping filter applied to each window.
<i>pTnsDirection</i>	As output. Pointer to the token that indicates whether the filter is applied in the upward or downward direction. 0 stands for upward and 1 for downward.
<i>pIcsInfo</i>	As input if <i>commonWin</i> == 1. As output if <i>commonWin</i> == 0. If <i>pIcsInfo->predDataPres</i> == 0, set <i>pIcsInfo->predReset</i> = 0. Only the first <i>pIcsInfo->numWinGrp</i> elements in <i>pIcsInfo->pWinGrpLen</i> are meaningful. Under specific conditions, some members of the structure must remain unchanged. See Table 10-5 .

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>pChanInfo</i> , <i>pDstScalefactor</i> , <i>pDstQuantizedSpectralCoef</i> , <i>pDstSfbCb</i> , <i>pDstTnsFiltCoef</i> , <i>pChanInfo->pIcsInfo</i> , or <i>*ppBitStream</i> is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of range [0, 7].
<code>ippStsAacComWinErr</code>	Indicates an error when <i>commonWin</i> exceeds [0, 1].
<code>ippStsAacSmplRateIdxErr</code>	Indicates an error when <i>pChanInfo->samplingRateIndex</i> exceeds [0, 11].
<code>ippStsAacPredSfbErr</code>	Indicates an error when <i>pChanInfo->predSfbMax</i> is not equal to 0.
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>pChanInfo->pIcsInfo->maxSfb</i> > <i>pChanInfo->numSwb</i> .
<code>ippStsAacSectCbErr</code>	Indicates an error when the codebook pointed to by <i>pChanInfo->pSectCb</i> is illegal or when (<i>pChanInfo->pSectCb</i>)==12, 13. If the current channel is not the right channel of the channel pair element, <i>pSectCb</i> = 14, 15 is also illegal.
<code>ippStsAacPlsDataErr</code>	Indicates an error when the <i>pChanInfo->pIcsInfo->winSequence</i> indicates a short sequence and <i>pChanInfo->pulsePres</i> indicates pulse data present. The start scale factor band for pulse data >= <i>pChanInfo->numSwb</i> , or pulse data position <i>offset</i> >= <i>winLen</i> .
<code>ippStsAacGainCtrErr</code>	Indicates an error when <i>pChanInfo->gainControlPres</i> is decoded as 1, which means that gain control data is present. Gain control data is not currently supported.
<code>ippStsAacCoefValErr</code>	Indicates an error when the quantized coefficients value pointed to by <i>pDstCoef</i> exceeds range [-8191, 8191].

DecodeDatStrElt_AAC

Gets data stream element from the input bitstream.

Syntax

```
IppStatus ippDecodeDatStrElt_AAC(Ipp8u** ppBitStream, int* pOffset, int*  
    pDataTag, int* pDataCnt, Ipp8u* pDstDataElt);
```

Parameters

<i>ppBitStream</i>	Double pointer to the current byte after the decode data stream element.
<i>pOffset</i>	Pointer to the bit position in the byte pointed to by <i>*ppBitStream</i> . Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
<i>pDataTag</i>	Pointer to <i>element_instance_tag</i> . See Table 6.20 of ISO/IEC 13818-7:1997 .
<i>pDataCn</i>	Pointer to the value of data length in bytes.
<i>pDstDataElt</i>	Pointer to the data stream buffer that contains the data stream extracted from the input bitstream. There are 512 elements in the buffer pointed to by <i>pDstDataElt</i> .

Description

This function is declared in the `ippac.h` file. The function gets data stream element from the input bitstream.

See clause 8.6 and Table 6.20 of [ISO/IEC 13818-7:1997](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>*ppBitStream</i> , <i>pDataTag</i> , <i>pDataCnt</i> , or <i>pDstDataElt</i> is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>*pOffset</i> is out of range [0, 7].

DecodeFillElt_AAC

Gets the fill element from the input bitstream.

Syntax

```
IppStatus ippDecodeFillElt_AAC(Ipp8u** ppBitStream, int* pOffset, int* pFillCnt,
    Ipp8u* pDstFillElt);
```

Parameters

<i>ppBitStream</i>	Pointer to the pointer to the current byte after the decode fill element.
<i>pOffset</i>	Pointer to the bit position in the byte pointed to by <i>ppBitStream</i> . Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
<i>pFillCnt</i>	Pointer to the value of the length of total fill data in bytes.
<i>pDstFillElt</i>	Pointer to the fill data buffer whose length must be equal to or greater than 270.

Description

This function is declared in the `ippac.h` file. The function gets the fill element from the input bitstream.

See clause 8.7 and Table 6.22 of [ISO/IEC 13818-7:1997](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>pFillCnt</i> , or <i>pDstFillElt</i> is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of the range of [0, 7].

QuantInv_AAC

Performs inverse quantization of Huffman symbols for current channel in-place.

Syntax

```
IppStatus ippsQuantInv_AAC_32s_I(Ipp32s* pSrcDstSpectralCoef, const Ipp16s*
    pScalefactor, int numWinGrp, const int* pWinGrpLen, int maxSfb, const Ipp8u*
    pSfbCb, int samplingRateIndex, int winLen);
```

Parameters

<i>pSrcDstSpectralCoef</i>	On input, pointer to the input quantized coefficients. For short block the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024. On output, pointer to the destination inverse quantized coefficient in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group.
<i>pScalefactor</i>	Buffer length must be more than or equal to 1024. The maximum error of output <i>pSrcDstSpectralCoef[i]</i> is listed in Table 10-7 . Pointer to the scalefactor buffer. Buffer length must be more than or equal to 120.
<i>numWinGrp</i>	Group number.
<i>pWinGrpLen</i>	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
<i>maxSfb</i>	Maximal scale factor bands number for the current block.
<i>pSfbCb</i>	Pointer to the scale factor band codebook, buffer length must be more than or equal to 120. Only <i>maxSfb</i> elements for each group are meaningful. There are no spaces between the sequence groups.
<i>samplingRateIndex</i>	Sampling rate index. Valid within [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997 .

winLen Data number in one window.

Description

This function is declared in the `ippac.h` file. The function performs inverse quantization of Huffman symbols for the current channel as shown by the formula.

$$pSrcDst[i] = \text{sign}(pSrcDst[i]) * (pSrcDst[i])^{\frac{4}{3}} * 2^{\left(\frac{1}{2}(pScalefactor[sfb] - 100)\right)}$$

See clause 10 of [ISO/IEC 13818-7:1997](#).

Table 10-7 Computation Error List for *pSrcDstSpectralCoef*

Output	Conditions	
<code>max(error (<i>pSrcDstSpectralCoef</i> [i]))</code>	Input abs (<i>pSrcDstSpectralCoef</i> [i])	Output abs (<i>pSrcDstSpectralCoef</i> [i])
3	<= 128	< 2 ^ 29
3	129~8191	<= 2 ^ 25
7	129~8191	< 2 ^ 29

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacSmplRateIdxErr</code>	Indicates an error when <i>samplingRateIndex</i> exceeds [0, 11].
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>maxSfb</i> exceeds [0, IPP_AAC_MAX_SFB].
<code>ippStsAacWinGrpErr</code>	Indicates an error when <i>numWinGrp</i> exceeds [0, 8] for long window or is not equal to 1 for short window.
<code>ippStsAacWinLenErr</code>	Indicates an error when <i>winLen</i> is not equal to 128 or 1024;
<code>ippStsAacCoefValErr</code>	Indicates an error when the quantized coefficients value pointed to by <i>pSrcDstSpectralCoef</i> exceeds [-8191, 8191].

DecodeMsStereo_AAC

Processes mid-side (MS) stereo for pair channels in-place.

Syntax

```
IppStatus ippsDecodeMsStereo_AAC_32s_I(Ipp32s* pSrcDstL, Ipp32s* pSrcDstR, int
    msMaskPres, const Ipp8u* pMsUsed, Ipp8u* pSfbCb, int numWinGrp, const int*
    pWinGrpLen, int maxSfb, int samplingRateIndex, int winLen);
```

Parameters

<i>pSrcDstL</i>	<p>On input, pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.</p> <p>On output, pointer to left channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.</p>
<i>pSrcDstR</i>	<p>On input, pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.</p> <p>On output, pointer to right channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.</p>
<i>msMaskPres</i>	<p>MS stereo mask flag:</p> <p>0 - MS off</p> <p>1 - MS on</p> <p>2 - MS all bands on.</p>
<i>pMsUsed</i>	<p>Pointer to the MS Stereo flag buffer. Buffer length must be more than or equal to 120.</p>
<i>pSfbCb</i>	<p>Pointer to the scale factor band codebook.</p> <p>If <code>invert_intensity (group, sfb) = -1</code>, and if <code>*pSfbCb = INTERITY_HCB</code>, let <code>*pSfbCb = INTERITY_HCB2</code>.</p>

If **pSfbCb* = INTERITY_HCB2, let
**pSfbCb* = INTERITY_HCB.

Buffer length must be more than or equal to 120. Store *maxSfb* elements for each group. There is no space between the sequence groups.

<i>numWinGrp</i>	Group number.
<i>pWinGrpLen</i>	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
<i>maxSfb</i>	Maximal scale factor bands number for the current block.
<i>samplingRateIndex</i>	Sampling rate index. Valid within [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997 .
<i>winLen</i>	Data number in one window.

Description

This function is declared in the `ippac.h` file. The function performs mid-side (MS) stereo process for pair channels and at the same time runs the `invert_intensity(group, sfb)` function and stores the values in the *pSfbCb* buffer.

In the case when MS stereo flag is on, perform operation on *pSrcDstL[i]* and *pSrcDstR[i]* as described in the following formula:

$$pSrcDstL'[i] = pSrcDstL[i] + pSrcDstR[i]$$

$$pSrcDstR'[i] = pSrcDstL[i] - pSrcDstR[i].$$

See clause 12 of [ISO/IEC 13818-7:1997](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacMaxSfbErr</code>	Indicates an error when the coefficient index calculated from <i>samplingFreqIndex</i> and <i>maxSfb</i> exceeds <i>winLen</i> in each window.
<code>ippStsAacSamplRateIdxErr</code>	Indicates an error when <i>pChanInfor->samplingRateIndex</i> exceeds [0, 11].

<code>ippStsAacWinGrpErr</code>	Indicates an error when <i>numWinGrp</i> exceeds [0, 8] for long window or is not equal to 1 for short window.
<code>ippStsAacWinLenErr</code>	Indicates an error when <i>winLen</i> is not equal 128 or 1024.
<code>ippStsStereoMaskErr</code>	Indicates an error when the stereo mask flag is not equal 1 or 2.

DecodelsStereo_AAC

Processes intensity stereo for pair channels.

Syntax

```
IppStatus ippDecodeIsStereo_AAC_32s(const Ipp32s* pSrcL, Ipp32s* pDstR,
    const Ipp16s* pScalefactor, const Ipp8u* pSfbCb, int numWinGrp,
    const int* pWinGrpLen, int maxSfb, int samplingRateIndex, int winLen);
```

Parameters

<i>pSrcL</i>	Pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
<i>pScalefactor</i>	Pointer to the scalefactor buffer. Buffer length must be more than or equal to 120.
<i>pSfbCb</i>	Pointer to the scale factor band codebook, buffer length must be more than or equal to 120. Store <i>maxSfb</i> elements for each group. There is no space between the sequence groups. Respective values of <i>pSfbCb[sfb]</i> equal to 1, -1, or 0 indicate the intensity stereo mode, that is, direct, inverse, or none.
<i>numWinGrp</i>	Group number.
<i>pWinGrpLen</i>	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
<i>maxSfb</i>	Maximal scalefactor bands number for the current block.
<i>samplingRateIndex</i>	Sampling rate index. Valid within [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997 .
<i>winLen</i>	Data number in one window.

pDstR Pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.

Description

This function is declared in the `ippac.h` file. The function processes intensity stereo for pair channels.

You should perform operation on *pSrcL[i]*, *pDstR[i]* according to the following formula.

$$pDstR[i] = pSrc[i] * is_intensity(g, sfb) * 2^{\left(-\frac{1}{4} pScalefactor[sfb]\right)}$$



NOTE. `invert_intensity(g, sfb)` is not used in the formula, because it is already decoded and stored in `pSfbCb[sfb]` in the MS stereo process primitive. Refer to clause 12 of [ISO/IEC 13818-7:1997](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsAacMaxSfbErr</code>	Indicates an error when the coefficient index calculated from <i>samplingFreqIndex</i> and <i>maxSfb</i> exceeds <i>winLen</i> in each window.
<code>ippStsAacSamplRateIdxErr</code>	Indicates an error when <i>pChanInfor->samplingRateIndex</i> exceeds <code>[0, 11]</code> .
<code>ippStsAacWinGrpErr</code>	Indicates an error when <i>numWinGrp</i> exceeds <code>[0, 8]</code> for long window or is not equal to 1 for short window.
<code>ippStsAacWinLenErr</code>	Indicates an error when <i>winLen</i> is not equal to 128 or 1024.

DeinterleaveSpectrum_AAC

Deinterleaves the coefficients for short block.

Syntax

```
IppStatus ippSDeinterleaveSpectrum_AAC_32s(const Ipp32s* pSrc, Ipp32s* pDst,
    int numWinGrp, const int* pWinGrpLen, int maxSfb, int samplingRateIndex,
    int winLen);
```

Parameters

<i>pSrc</i>	Pointer to source coefficients buffer. The coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
<i>numWinGrp</i>	Group number.
<i>pWinGrpLen</i>	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
<i>maxSfb</i>	Maximal scalefactor bands number for the current block.
<i>samplingRateIndex</i>	Sampling rate index. Valid in [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997 .
<i>winLen</i>	Data number in one window.
<i>pDst</i>	Pointer to the output of coefficients. Data sequence is ordered in $pDst[w*128+sfb*sfbWidth[sfb]+i]$, where w is window index, sfb is scale factor band index, $sfbWidth$ is the scale factor band width table, i is the index within scale factor band. Buffer length must be more than or equal to 1024.

Description

This function is declared in the `ippac.h` file. The function deinterleaves the coefficients for short block.

See clause 8.3.5 of [ISO/IEC 13818-7:1997](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacMaxSfbErr</code>	Indicates an error when the coefficient index calculated from <i>samplingFreqIndex</i> and <i>maxSfb</i> exceeds <i>winLen</i> in each window.
<code>ippStsAacSamplRateIdxErr</code>	Indicates an error when <i>pChanInfor->samplingRateIndex</i> exceeds [0,11].
<code>ippStsAacWinGrpErr</code>	Indicates an error when <i>numWinGrp</i> exceeds [0,8].
<code>ippStsAacWinLenErr</code>	Indicates an error when <i>winLen</i> is not equal to 128.

DecodeTNS_AAC

Decodes for Temporal Noise Shaping in-place.

Syntax

```
IppStatus ippDecodeTNS_AAC_32s_I(Ipp32s* pSrcDstSpectralCoefs, const int*
    pTnsNumFilt, const int* pTnsRegionLen, const int* pTnsFiltOrder, const int*
    pTnsFiltCoefRes, const Ipp8s* pTnsFiltCoef, const int* pTnsDirection,
    int maxSfb, int profile, int samplingRateIndex, int winLen);
```

Parameters

<i>pSrcDstSpectralCoefs</i>	On input, pointer to the input spectral coefficients to be filtered by the all-pole filters in Q13.18 format. There are 1024 elements in the buffer pointed to by <i>pSrcDstSpectralCoefs</i> . On output, pointer to the output spectral coefficients after filtering by the all-pole filters in Q13.18 format. See Table 10-8 for the computation error compared with the double precision data.
<i>pTnsNumFilt</i>	Pointer to the number of noise shaping filters used for each window of the current frame. There are 8 elements in the buffer pointed to by <i>pTnsNumFilt</i> which are arranged as follows: <i>pTnsNumFilt[w]</i> : the number of noise shaping filters used for window <i>w</i> , <i>w</i> = 0 to <i>numWin</i> -1.
<i>pTnsRegionLen</i>	Pointer to the length of the region in units of scale factor bands to which one filter is applied in each window of the current frame.

There are 8 elements in the buffer pointed to by *pTnsRegionLen*, which are arranged as follows:

pTnsRegionLen[i]: the length of the region to which filter *filt* is applied in window *w*

$$i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt$$

w = 0 to *numWin*-1, *filt*=0 to *pTnsNumFilt*[*w*]-1.

pTnsFiltOrder

Pointer to the order of one noise shaping filter applied to each window of the current frame. There are 8 elements in the buffer pointed to by *pTnsFiltOrder*, which are arranged as follows:

pTnsFiltOrder[i]: the order of one noise shaping filter *filt*, which is applied to window *w*

$$i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt$$

w = 0 to *numWin*-1, *filt*=0 to *pTnsNumFilt*[*w*]-1.

pTnsFiltCoefRes

Pointer to the resolution of 3 bits or 4 bits of the transmitted filter coefficients for each window of the current frame. There are 8 elements in the buffer pointed to by *pTnsFiltCoefRes*, which are arranged as follows:

pTnsFiltCoefRes[w]: the resolution of the transmitted filter coefficients for window *w*, *w* = 0 to *numWin*-1.

pTnsFiltCoef

Pointer to the coefficients of one noise shaping filter applied to each window of the current frame. There are 60 elements in the buffer pointed to by *pTnsFiltCoef*, which are arranged as follows:

pTnsFiltCoef[i], *pTnsFiltCoef[i+1]*, ..., *pTnsFiltCoef[i+order-1]*: the coefficients of one noise shaping filter *filt*, which is applied to window *w*.

The order is the same as that of the noise shaping filter *filt* as applied to window *w*, *w* = 0 to *numWin*-1, *filt*=0 to *pTnsNumFilt*[*w*]-1.

For example, $pTnsFiltCoef[0]$, $pTnsFiltCoef[1]$, ..., $pTnsFiltCoef[order0-1]$ are the coefficients of the noise shaping filter 0, which is applied to window 0, if present.

If so, $pTnsFiltCoef[order0]$, $pTnsFiltCoef[order0+1]$, ..., $pTnsFiltCoef[order0+order1-1]$ are the coefficients of the noise shaping filter 1 applied to window 0, if present, and so on.

$order0$ is the same as that of the noise shaping filter 0 applied to window 0, and $order1$ is the order of the noise shaping filter 1 applied to window 0.

After window 0 is processed, process window 1, then window 2 until all $numWin$ windows are processed.

$pTnsDirection$

Pointer to the token that indicates whether the filter is applied in the upward or downward direction.

0 stands for upward and 1 for downward.

There are 8 elements in the buffer pointed to by $pTnsDirection$ which are arranged as follows:

$pTnsDirection[i]$: the token indicating whether the filter $filt$ is applied in upward or downward direction to window w

$$i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt$$

$w = 0$ to $numWin-1$, $filt=0$ to $pTnsNumFilt[w]-1$.

$maxSfb$

Number of scale factor bands transmitted per window group of the current frame.

$profile$

Profile index from Table 7.1 in [ISO/IEC 13818-7:1997](#).

$samplingRateIndex$

Index indicating the sampling rate of the current frame.

$winLen$

Data number in one window.

Description

This function is declared in the `ippac.h` file. The function performs decoding process for Temporal Noise Shaping (TNS) that controls the temporal shape of the quantization noise within each window of the transform.

The TNS decoding process proceeds separately for each window of the current frame by applying the all-pole filtering to selected regions of the spectral coefficients.

Table 10-8 Computation Error List for *pSrcDstSpectralCoefs*

MAX(error(<i>pSrcDstSpectralCoefs</i> [<i>i</i>]))	Condition
4095	8 == numWin
32767	1 == numWin

numWin is the number of windows in a window sequence of the current frame. *numWin* is equal to 8 if window sequence is EIGHT_SHORT_SEQUENCE, or to 1 for other window sequences.

numSwb is the total number of scale factor window bands for the actual window type, that is, long or short window of the current frame.



NOTE. This function supports LC profile only.

Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when at least one of the specified pointers is NULL.
<i>IppStsTnsProfileErr</i>	Indicates an error when <i>profile</i> != 1.
<i>ippStsAacTnsNumFiltErr</i>	Indicates an error when a data error occurs: for a short window sequence <i>pTnsNumFilt</i> [<i>w</i>] exceeds [0,1]; for a long window sequence, <i>pTnsNumFilt</i> [<i>w</i>] exceeds [0,3].
<i>ippStsAacTnsLenErr</i>	Indicates an error when <i>pTnsRegionLen</i> exceeds [0, <i>numSwb</i>].
<i>ippStsAacTnsOrderErr</i>	Indicates an error when a data error occurs: for a short window sequence, <i>pTnsFiltOrder</i> exceeds [0,7]; for a long window sequence, <i>pTnsFiltOrder</i> exceeds [0,12].

<code>ippStsAacTnsCoefResErr</code>	Indicates an error when <i>pTnsFiltCoefRes[w]</i> exceeds [3, 4].
<code>ippStsAacTnsCoefErr</code>	Indicates an error when <i>*pTnsFiltCoef</i> exceeds [-8, 7].
<code>ippStsAacTnsDirectErr</code>	Indicates an error when <i>*pTnsDirection</i> exceeds [0, 1].



NOTE. *numWin* is the number of windows in a window sequence of the current frame. *numWin* is 8 if window sequence is EIGHT_SHORT_SEQUENCE, or 1 for other window sequences.

NOTE. *numSwb* is the total number of scale factor window bands for the actual window type, that is, long or short window, of the current frame.

MDCTInv_AAC

Maps time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples.

Syntax

```
IppStatus ippMDCTInv_AAC_32s16s(Ipp32s* pSrcSpectralCoefs, Ipp16s*
    pDstPcmAudioOut, Ipp32s* pSrcDstOverlapAddBuf, int winSequence, int
    winShape, int prevWinShape, int pcmMode);
```

Parameters

<i>pSrcSpectralCoefs</i>	Pointer to the input time-frequency domain samples in Q13.18 format. There are 1024 elements in the buffer pointed to by <i>pSrcSpectralCoefs</i> .
<i>pSrcDstOverlapAddBuf</i>	Pointer to the overlap-add buffer that contains the second half of the previous block windowed sequence in Q13.18. There are 1024 elements in this buffer.

<i>winSequence</i>	Flag that indicates which window sequence is used for current block.
<i>winShape</i>	Flag that indicates which window function is selected for current block.
<i>prevWinShape</i>	Flag that indicates which window function is selected for previous block.
<i>pcmMode</i>	Flag that indicates whether the PCM audio output is interleaved, that is has the pattern LRLRLR... or not. 1 stands for not interleaved. 2 stands for interleaved
<i>pDstPcmAudioOut</i>	Pointer to the output 1024 reconstructed 16-bit signed little-endian PCM samples in Q15, interleaved, if needed. The maximum computation error for <i>pDstPcmAudioOut</i> is less than 1 for each vector element. The total quadratic error for the vector is less than 96.
<i>pSrcDstOverlapAddBuf</i>	Pointer to the overlap-add buffer which contains the second half of the current block windowed sequence in Q13.18. The maximum computation error for <i>pDstPcmAudioOut</i> is less than 4 for each vector element. The total quadratic error for the vector is less than 1536.

Description

This function is declared in the `ippac.h` file. This function maps the time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples as output for each channel.

This module consists of

- IMDCT transform
- windowing
- overlap-add operation.

In order to adapt the time/frequency resolution of the filterbank to the characteristics of the input signal, a block switching tool is also adopted. For each channel, 1024 time-frequency domain samples are transformed into the time domain via the IMDCT.

After applying the windowing operation, the first half of the windowed sequence is added to the second half of the previous block windowed sequence to reconstruct 1024 output samples for each channel. Output can be interleaved according to *pcmMode*.

If *pcmMode* equals 2, the output is in the sequence *pDstPcmAudioOut[2*i]*, *i*=0 to 1023, that is, 1024 output samples are stored in the sequence: *pDstPcmAudioOut[0]*, *pDstPcmAudioOut[2]*, *pDstPcmAudioOut[4]*,..., *pDstPcmAudioOut[2046]*.

If *pcmMode* equals 1, the output is in the sequence *pDstPcmAudioOut[i]*, *i*=0 to 1023.

You should also preallocate an input-output buffer pointed to by *pSrcDstOverlapAddBuf* for the overlap-add operation.

Reset this buffer to zero before the first call and then use the output of the current call as the input of the next call for the same channel.

Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when at least one of the specified pointers is NULL.
<i>ippStsAacWinSeqErr</i>	Indicates an error when <i>winSequence</i> exceeds [0,3].
<i>ippStsAacWinShapeErr</i>	Indicates an error when <i>winShape</i> or <i>prevWinShape</i> exceeds [0,1].
<i>ippStsAacPcmModeErr</i>	Indicates an error when <i>pcmMode</i> exceeds [1,2].

MPEG-4 AAC Primitives

This section introduces primitive functions for MPEG-4 AAC operations.

DecodeMainHeader_AAC

Gets main header information and main layer information from bit stream.

Syntax

```
IppStatus ippDecodeMainHeader_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACMainHeader* pAACMainHeader, int channelNum, int monoStereoFlag);
```

Parameters

<i>ppBitStream</i>	Double pointer to bitstream buffer. <i>*ppBitStream</i> will be updated after decoding.
<i>pOffset</i>	Pointer to the offset in one byte. <i>*pOffset</i> will be updated after decoding.
<i>channelNum</i>	Number of channels.
<i>monoStereoFlag</i>	Current frame has mono and stereo layers.
<i>pAACMainHeader</i>	Pointer to the main element header.

Description

This function is declared in the `ippac.h` file. The function gets main header information and main layer information from bit stream.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> exceeds <code>[0, 7]</code> .
<code>ippStsAacChanErr</code>	Indicates an error when <i>channelNum</i> exceeds <code>[1, 2]</code> .
<code>ippStsAacMonoStereoErr</code>	Indicates an error when <i>monoStereoFlag</i> exceeds <code>[0, 1]</code> .

DecodeExtensionHeader_AAC

Gets extension header information and extension layer information from bit stream.

Syntax

```
IppStatus ippDecodeExtensionHeader_AAC(Ipp8u** ppBitStream, int* pOffset,  
    IppAACExtHeader* pAACExtHeader, int monoStereoFlag, int thisLayerStereo, int  
    monoLayerFlag, int preStereoMaxSfb, int highstMonoMaxSfb, int winSequence);
```

Parameters

<i>ppBitStream</i>	Double pointer to bitstream buffer. <i>*ppBitStream</i> will be updated after decoding.
<i>pOffset</i>	Pointer to the offset in one byte. <i>*pOffset</i> will be updated after decoding.
<i>monoStereoFlag</i>	Flag indicating that the current frame has mono and stereo layers.
<i>thisLayerStereo</i>	Flag indicating that the current layer is stereo.
<i>monoLayerFlag</i>	Flag indicating that the current frame has a mono layer.
<i>preStereoMaxSfb</i>	Previous stereo layer <i>maxSfb</i> .
<i>hightstMonoMaxSfb</i>	Last mono layer <i>maxSfb</i> .
<i>winSequence</i>	Window type, short or long.
<i>pAACExtHeader</i>	Pointer to the extension element header.

Description

This function is declared in the `ippac.h` file. The function gets extension header information and extension layer information from the bitstream.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of the range of <code>[0, 7]</code> .
<code>ippStsAacStereoLayerErr</code>	Indicates an error when <i>thisLayerStereo</i> exceeds <code>[0, 1]</code> .
<code>ippStsAacMonoLayerErr</code>	Indicates an error when <i>monoLayerFlag</i> exceeds <code>[0, 1]</code> .
<code>ippStsAacMaxSfbErr</code>	Indicates an error when at least one of the parameters <i>preStereoMaxSfb</i> , <i>hightstMonoMaxSfb</i> or the number of scale factor bands used in this layer exceeds <code>[0, IPP_AAC_MAX_SFB]</code> .
<code>ippStsAacMonoStereoErr</code>	Indicates an error when <i>monoStereoFlag</i> exceeds <code>[0, 1]</code> .
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds <code>[0, 3]</code> .

DecodePNS_AAC

Implements perceptual noise substitution (PNS) coding within individual channel stream (ICS).

Syntax

```
IppStatus ippDecodePNS_AAC_32s(Ipp32s* pSrcDstSpec, int* pSrcDstLtpFlag, Ipp8u*
    pSfbCb, Ipp16s* pScaleFactor, int maxSfb, int numWinGrp, int* pWinGrpLen,
    int samplingFreqIndex, int winLen, int* pRandomSeed);
```

Parameters

<i>pSrcDstSpec</i>	Pointer to spectrum coefficients for perceptual noise substitution (PNS).
<i>pSrcDstLtpFlag</i>	Pointer to long term predict (LTP) flag.
<i>pSfbCb</i>	Pointer to the scale factor codebook.
<i>pScaleFactor</i>	Pointer to the scalefactor value.
<i>maxSfb</i>	Number of scale factor bands used in this layer.
<i>numWinGrp</i>	Number of window groups.
<i>pWinGrpLen</i>	Pointer to the length of every window group.
<i>samplingFreqIndex</i>	Sampling frequency index.
<i>winLen</i>	Window length. 1024 for long windows, 128 for short windows.
<i>pRandomSeed</i>	Random seed for PNS.

Description

This function is declared in the `ippac.h` file. The function implements perceptual noise substitution (PNS) coding within the individual channel stream (ICS). Certain sets of spectral coefficients are derived from random vectors rather than from Huffman-coded symbols and inverse quantization process.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>maxSfb</i> exceeds <code>[0, IPP_AAC_MAX_SFB]</code> .
<code>ippStsAacSmplRateIdxErr</code>	Indicates an error when <i>samplingFreqIndex</i> exceeds <code>[0,16]</code> .
<code>ippStsAacWinLenErr</code>	Indicates an error when <i>winLen</i> is not equal to 128 or 1024.

DecodeMsPNS_AAC

Implements perceptual noise substitution (PNS) coding in the case of joint coding.

Syntax

```
IppStatus ippsDecodeMsPNS_AAC_32s( Ipp32s* pSrcDstSpec, int* pSrcDstLtpFlag, Ipp8u*
    pSfbCb, Ipp16s* pScaleFactor, int maxSfb, int numWinGrp, int* pWinGrpLen,
    int samplingFreqIndex, int winLen, int* pRandomSeed, int channel, Ipp8u* pMsUsed,
    int* pNoiseState);
```

Parameters

<i>pSrcDstSpec</i>	Pointer to spectrum coefficients for perceptual noise substitution (PNS).
<i>pSrcDstLtpFlag</i>	Pointer to long term predict (LTP) flag.
<i>pSfbCb</i>	Pointer to the scalefactor code book.
<i>pScaleFactor</i>	Pointer to the scalefactor coefficients.
<i>maxSfb</i>	Number of maximum scalefactor band.
<i>numWinGrp</i>	Number of window groups.
<i>pWinGrpLen</i>	Pointer to the length of every window group.
<i>samplingFreqIndex</i>	Sampling frequency index.
<i>winLen</i>	Window length.
<i>pRandomSeed</i>	Random seed.
<i>channel</i>	Index of the current channel: 0 means left, 1 means right.
<i>pMsUsed</i>	Pointer to MS-used buffer in CPE structure.

pNoiseState Pointer to the noise state buffer, which stores the left channel noise random seed for every scalefactor band. When *pMsUsed[sfb]==1*, the content in this buffer is used for the right channel.

Description

This function is declared in the `ippac.h` file. The function implements perceptual noise substitution (PNS) coding in the case of joint coding. Certain sets of spectral coefficients are derived from random vectors rather than from Huffman-coded symbols and inverse quantization process.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>maxSfb</i> exceeds <code>[0, IPP_AAC_MAX_SFB]</code> .
<code>ippStsAacWinGrpErr</code>	Indicates an error when <i>maxWinGrp</i> exceeds <code>[0,8]</code> .
<code>ippStsAacSmplRateIdxErr</code>	Indicates an error when <i>samplingFreqIndex</i> exceeds <code>[0,16]</code> .
<code>ippStsAacWinLenErr</code>	Indicates an error when <i>winLen</i> is not equal to 128 or 1024.

DecodeChanPairElt_MP4_AAC

Gets channel_pair_element from the input bitstream.

Syntax

```
IppStatus ippDecodeChanPairElt_MP4_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACIcsInfo* pIcsInfo, IppAACChanPairElt* pChanPairElt, IppAACMainHeader*
    pAACMainHeader, int predSfbMax, int audioObjectType);
```

Parameters

<i>ppBitStream</i>	Double pointer to the current byte. <i>*ppBitStream</i> will be updated after decoding.
<i>pOffset</i>	Pointer to the bit position in the byte pointed to by <i>*ppBitStream</i> . Valid within 0 to 7: 0 stands for the most significant bit of the byte; 7 stands for the least significant bit of the byte.
<i>pIcsInfo</i>	Pointer to <code>IppAACIcsInfo</code> structure.
<i>pChanPairElt</i>	Pointer to <code>IppAACChanPairElt</code> structure.
<i>pAACMainHeader</i>	Pointer to the main element header.
<i>predSfbMax</i>	Maximum prediction scale factor bands.
<i>audioObjectType</i>	Audio object type indicator: 1 indicates the main type 2 indicates the LC type 6 indicates the scalable mode.

Description

The `ippsDecodeChanPairElt_MP4_AAC` function is declared in the `ippac.h` file. This function gets the channel pair element from the input bitstream. Individual channel stream is not included.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of the range of <code>[0, 7]</code> .
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>pIcsInfo->maxSfb</i> decoded from bitstream is greater than <code>IPP_AAC_MAX_SFB</code> , the maximum scale factor band for all sampling frequencies.

LongTermReconstruct_AAC

Uses Long Term Reconstruct (LTR) to reduce signal redundancy between successive coding frames.

Syntax

```
IppStatusippsLongTermReconstruct_AAC_32s(Ipp32s*pSrcEstSpec,Ipp32s*pSrcDstSpec,  
int* pLtpFlag, int winSequence, int samplingFreqIndex);
```

Parameters

<i>pSrcDstSpec</i>	Pointer to spectral coefficients for LTP.
<i>pSrcEstSpec</i>	Pointer to the frequency domain vector.
<i>winSequence</i>	Window type, long or short.
<i>samplingFreqIndex</i>	Sampling frequency index.
<i>pLtpFlag</i>	Pointer to the LTP flag.

Description

This function is declared in the `ippac.h` file. The function uses Long Term Reconstruct (LTR) to reduce signal redundancy between successive coding frames.

LTP is a forward adaptive predictor that is inherently less sensitive to the round-off numerical errors in the decoder or bi-errors in the transmitted spectral coefficients.

You should add the vector of decoded spectral coefficients and the corresponding frequency domain vector to get the vector of reconstructed spectral coefficients.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacSmplRateIdxErr</code>	Indicates an error when <i>samplingFreqIndex</i> exceeds <code>[0,12]</code> .
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds <code>[0,3]</code> .

MDCTFwd_AAC

Generates spectrum coefficient of PCM samples.

Syntax

```
IppStatus ippMDCTFwd_AAC_32s(Ipp32s* pSrc, Ipp32s* pDst, Ipp32s*
    pSrcDstOverlapAdd, int winSequence, int winShape, int preWinShape, Ipp32s*
    pWindowedBuf);
```

Parameters

<i>pSrc</i>	Pointer to temporal signals to do MDCT.
<i>pDst</i>	Output of MDCT, the spectral coefficients of PCM samples.
<i>pSrcDstOverlapAdd</i>	Pointer to overlap buffer. Not used for MPEG-4 AAC decoding.
<i>winSequence</i>	Window sequence indicating if the block is long or short.
<i>winShape</i>	Current window shape.
<i>preWinShape</i>	Previous window shape.
<i>pWindowedBuf</i>	Work buffer for MDCT. Should be at least 2048 words.

Description

This function is declared in the `ippac.h` file. The function generates the spectrum coefficient of PCM samples in the MDCT Long Term Reconstruct (LTP) loop.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds <code>[0, 3]</code> .
<code>ippStsAacWinShapeErr</code>	Indicates an error when <i>preWinShape</i> exceeds <code>[0, 1]</code> .

EncodeTNS_AAC

*Performs reversion of TNS in the Long Term
Reconstruct loop in-place.*

Syntax

```
IppStatus ippEncodeTNS_AAC_32s_I(Ipp32s* pSrcDst, const int* pTnsNumFilt,  
    const int* pTnsRegionLen, const int* pTnsFiltOrder, const int*  
    pTnsFiltCoefRes, const Ipp8s* pTnsFiltCoef, const int* pTnsDirection, int  
    maxSfb, int profile, int samplingRateIndex, int winLen);
```

Parameters

<i>pSrcDst</i>	On input, pointer to the spectral coefficients for the TNS encoding operation. On output, pointer to the spectral coefficients after the TNS encoding operation.
<i>pTnsNumFilt</i>	Pointer to the number of TNS filters.
<i>pTnsRegionLen</i>	Pointer to the length of TNS filter.
<i>pTnsFiltOrder</i>	Pointer to the TNS filter order.
<i>pTnsFiltCoefRes</i>	Pointer to the TNS coefficient resolution flag.
<i>pTnsFiltCoef</i>	Pointer to the TNS filter coefficients.
<i>pTnsDirection</i>	Pointer to the TNS direction flag.
<i>maxSfb</i>	Maximum scale factor number.
<i>profile</i>	Audio profile.
<i>samplingRateIndex</i>	Sampling frequency index.
<i>winLen</i>	Window length.

Description

This function is declared in the `ippac.h` file. The function performs in-place reversion of TNS in the LTP loop, or Analysis Temporal Noise Shaping.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsTnsProfileErr</code>	Indicates an error when <code>profile != 1</code> .
<code>ippStsAacTnsNumFiltErr</code>	Indicates an error when <code>*pTnsNumFilt</code> exceeds <code>[0, 1]</code> for the short window sequence or <code>[0, 3]</code> for the long window sequence.
<code>ippStsAacTnsLenErr</code>	Indicates an error when <code>*pTnsRegionLen</code> exceeds <code>[0, numSwb]</code> .
<code>ippStsAacTnsOrderErr</code>	Indicates an error when <code>*pTnsFiltOrder</code> exceeds <code>[0, 7]</code> for the short window sequence or <code>[0, 12]</code> for the long window sequence.
<code>ippStsAacTnsCoefResErr</code>	Indicates an error when <code>*pTnsFiltCoefRes</code> exceeds <code>[3, 4]</code> .
<code>ippStsAacTnsCoefErr</code>	Indicates an error when <code>*pTnsFiltCoef</code> exceeds <code>[-8, 7]</code> .
<code>ippStsAacTnsDirectErr</code>	Indicates an error when <code>*pTnsDirection</code> exceeds <code>[0, 1]</code> .
<code>ippStsAacSmplRateIdxErr</code>	Indicates an error when <code>samplingRateIndex</code> exceeds <code>[0, 11]</code> .
<code>ippStsAacWinLenErr</code>	Indicates an error when <code>winLen</code> is not equal to 128 or 1024.

LongTermPredict_AAC

Gets the predicted time domain signals in the Long Term Reconstruct (LTP) loop.

Syntax

```
IppStatus ippLongTermPredict_AAC_32s(Ipp32s* pSrcTimeSignal, Ipp32s*
    pDstEstTimeSignal, IppAACLtpInfo* pAACLtpInfo, int winSequence);
```

Parameters

<i>pSrcTimeSignal</i>	Pointer to the temporal signals to be predicted in the temporary domain.
<i>pDstEstTimeSignal</i>	Pointer to the output of samples after LTP.
<i>pAACLtpInfo</i>	Pointer to the LTP information.
<i>winSequence</i>	Window type, short or long.
<i>pDstEstTimeSignal</i>	Pointer to the prediction output in time domain.

Description

This function is declared in the `ippac.h` file. The function gets the predicted time domain signals in the LTP loop.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers is <code>NULL</code> .
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds <code>[0,3]</code> .

NoiselessDecode_AAC

Performs noiseless decoding.

Syntax

```
IppStatus ippNoiselessDecode_AAC(Ipp8u** ppBitStream, int* pOffset,  
    IppAACMainHeader* pAACMainHeader, Ipp16s* pDstScaleFactor, Ipp32s*  
    pDstQuantizedSpectralCoef, Ipp8u* pDstSfbCb, Ipp8s* pDstTnsFiltCoef,  
    IppAACChanInfo* pChanInfo, int winSequence, int maxSfb, int commonWin, int  
    scaleFlag, int audioObjectType);
```

Parameters

<i>ppBitStream</i>	Double pointer to the bitstream to be parsed. <i>*ppBitStream</i> is be updated after decoding.
<i>pOffset</i>	Pointer to the offset in one byte. <i>*pOffset</i> is be updated after decoding.

<i>pAACMainHeader</i>	Pointer to main header information. Not used for scalable objects. When <i>commonWin == 0</i> && <i>scaleFlag==0</i> , you need to decode LTP information and save it in <i>pAACMainHeader->pLtpInfo[]</i> .
<i>pChanInfo</i>	Pointer to channel information structure.
<i>windowSequence</i>	Window type, short or long.
<i>maxSfb</i>	Number of scale factor bands.
<i>commonWin</i>	Indicates if the channel pair uses the same ICS information.
<i>scaleFlag</i>	Flag indicating whether the scalable type is used.
<i>audioObjectType</i>	Audio object type indicator: 1 indicates the main type 2 indicates the LC type 6 indicates the scalable mode.
<i>pDstScaleFactor</i>	Pointer to the parsed scalefactor.
<i>pDstQuantizedSpectralCoef</i>	Pointer to the quantized spectral coefficients after Huffman decoding.
<i>pDstSfbCb</i>	Pointer to the scale factor codebook index.
<i>pDstTnsFiltCoef</i>	Pointer to TNS filter coefficients. Not used for scalable objects.

Description

This function is declared in the `ippac.h` file. This is a general noiseless decoding module for MPEG-2 and MPEG-4 objects.

In case one scale factor band uses PNS in MPEG-4 AAC scalable object, **pDstScaleFactor* contains the noise energy for this scale factor band and *pDstSfbCb[sfb]* should be `NOISE_HCB(13)`. The spectrum in this scale factor band is not necessarily Huffman-decoded, and the *pDstQuantizedSpectralCoef* of this scale factor band can be zero.

In AAC scalable object, *pDstTnsFiltCoef* and *pAACMainHeader* are not used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <code>pOffset</code> is out of the range <code>[0,7]</code> .
<code>ippStsAacComWinErr</code>	Indicates an error when <code>commonWin</code> exceeds <code>[0,1]</code> .
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <code>maxSfb</code> exceeds <code>[0,IPP_AAC_MAX_SFB]</code> .
<code>ippStsAacSmplRateIdxErr</code>	Indicates an error when <code>pChanInfo->samplingRateIndex</code> exceeds <code>[0,11]</code> .
<code>ippStsAacCoefValErr</code>	Indicates an error when the quantized coefficients value pointed to by <code>pDstCoef</code> exceeds the range of <code>[-8191,8191]</code> .

LtpUpdate_AAC

Performs required buffer update in the Long Term Reconstruct (LTP) loop.

Syntax

```
IppStatus ippStsLtpUpdate_AAC_32s(Ipp32s* pSpecVal, Ipp32s* pLtpSaveBuf, int winSequence, int winShape, int preWinShape, Ipp32s* pWorkBuf);
```

Parameters

<code>pSpecVal</code>	Pointer to spectral value after TNS decoder in LTP loop.
<code>pLtpSaveBuf</code>	Pointer to save buffer for LTP. Buffer length should be <code>3*frameLength</code> . The value is saved for the next frame.
<code>winSequence</code>	Window type: 0 stands for long 1 stands for long start 2 stands for short 3 stands for long stop.
<code>winShape</code>	KBD or SIN window shape.
<code>preWinShape</code>	Previous window shape.

pWorkBuf Work buffer for LTP update, length of *pWorkBuf* should be at least $2048 * 3 = 6144$ words.

Description

This function is declared in the `ippac.h` file. The function performs required buffer update in the Long Term Reconstruct (LTP) loop. This operation includes IMDCT and updating the save buffer.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds <code>[0, 3]</code> .
<code>ippStsAacWinShapeErr</code>	Indicates an error when <i>winShape</i> or <i>preWinShape</i> exceeds <code>[0, 1]</code> .

Spectral Band Replication Functions

SBR (Spectral Band Replication) is a new audio coding bandwidth extension technology, which is standardized in ISO/IEC 14496-3:2001/Amd.1:2003 [[ISO14496A](#)].

It improves the performance of low bitrate audio and speech codecs by either increasing the audio bandwidth at a given bitrate or by improving coding efficiency at a given quality level. For example, SBR can be used in conjunction with mp3 (mp3PRO), AAC (HE-AAC), CELP, HVXC to achieve a quality at 64 kbps stereo (mp3, AAC) that compares to conventional stereo at a bitrate exceeding 100 kbps.

SBR can be used with mono and stereo as well as with multichannel audio.

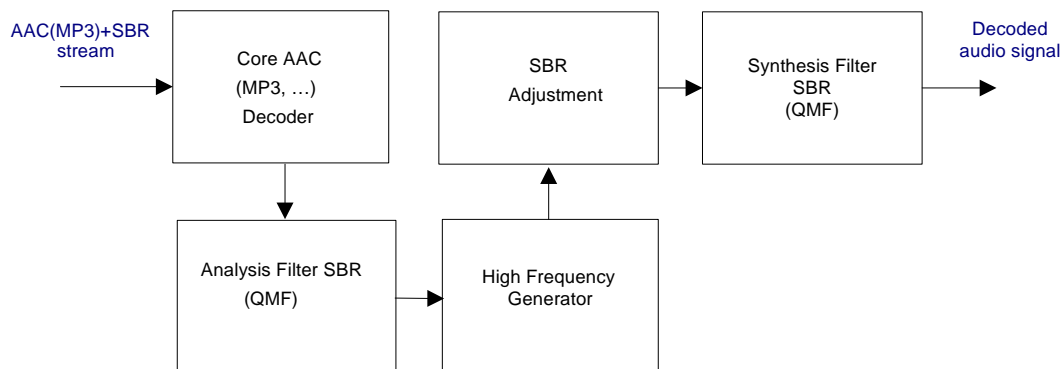
HE-AAC profile is combining AAC LC profile and SBR technology. HE-AAC is widely for mobile multimedia services.

SBR technology can operate on complex-valued signals in the mode called High Quality (HQ) or on real-valued signals in the mode called Low Power (LP)).

In accordance with the mathematical model used by SBR in most audio materials a very high correlation is observed between the lower frequencies and the higher frequencies of a spectrum. Therefore “smart” transposition allows the transmitted signals in higher frequencies to be restructured so as to avoid coding and transmitting them as spectral data. The subsequent SBR adjustment block improves the highest reconstructed frequency.

[Figure 10-4](#) shows the sequence blocks of SBR decoder tool.

Figure 10-4 SBR Pipeline



QMF Functions

A Quadrature Mirror Filter (QMF) bank is used by MPEG-4 SBR decoder. QMF consists of

- Analysis Filter
- Synthesis Filter
- Synthesis Down Filter.
- Analysis Filter bank is used to split the time domain signal output from the core decoder into 32 subband signals. The output from the filterbank, that is, the subband samples, is complex-valued for HQ SBR mode or real-valued for LP SBR mode.
- Synthesis Filter bank is used to filter SBR-processed subband signals. The output from the filterbank is real-valued time domain samples. Filtering is achieved using a 64-subband Filter bank.
- Down Synthesis Filter bank is used to filter SBR-processed subband signals. The output from the filterbank is real-valued time domain samples. Filtering is achieved using a 32-subband Filter bank.

QMF filterbanks use the functions given below in low power (LP) and high quality (HQ) modes. Each mode is characterized by three function groups:

- AnalysisFilter
- SynthesisFilter
- SynthesisDownFilter.

Each group has functions for initialization and size getting. The general operational algorithm is as follows:

1. Call `ipps(**)FilterGetSize_SBR_(***)`, where `(**)` is a function group name, that is `Analysis`, `Synthesis` or `SynthesisDown`, and `(***)` is a flavor descriptor (for example, `_RToC_32f32fc`, `_RToR_32f`, etc.)
2. Allocate memory (external operation)
3. Call `ipps(**)FilterInit_SBR_(***)` to initialize the corresponding main function
4. Use `ipps(**)Filter_SBR_(***)` for the main operation.
5. Deallocate the memory (external operation).

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm> .

Size Getting

These functions return the size in bytes of `FilterSpec` structures, the buffer for initialization process, and the work buffer.

AnalysisFilterGetSize_SBR

Returns size of `FilterSpec_SBR` specification structures, init and work buffers.

Syntax

```
IppStatus ippAnalysisFilterGetSize_SBR_RToC_32f32fc(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
IppStatus ippAnalysisFilterGetSize_SBR_RToC_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
IppStatus ippAnalysisFilterGetSize_SBR_RToR_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
```

Parameters

<i>pSizeSpec</i>	Address of the <code>FilterSpec_SBR</code> specification structure size value in bytes.
<i>pSizeInitBuf</i>	Address size of the buffer for initialization functions in bytes.
<i>pSizeWorkBuf</i>	Address of the work buffer size value in bytes.

Description

These functions are declared in the `ippac.h` file. The functions return sizes of the work buffer, specification structure and initialization buffer for `Init` functions for Analysis Filter bank.

`ippAnalysisFilterGetSize_SBR_RToC_32f32fc` returns the specified sizes for [ippAnalysisFilter_SBR_RToC_32f32fc_D2L](#).

`ippAnalysisFilterGetSize_SBR_RToC_32f` returns the specified sizes for [ippAnalysisFilter_SBR_RToC_32f_D2L](#).

`ippAnalysisFilterGetSize_SBR_RToR_32f` returns the specified sizes for [ippAnalysisFilter_SBR_RToR_32f_D2L](#).

The functions `ippsAnalysisFilterGetSize_SBR_RToC_32f` and `ippsAnalysisFilterGetSize_SBR_RToR_32f` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

SynthesisFilterGetSize_SBR

Returns size of `FilterSpec_SBR` specification structures, init and work buffers.

Syntax

```
IppStatus ippsSynthesisFilterGetSize_SBR_CToR_32fc32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
IppStatus ippsSynthesisFilterGetSize_SBR_CToR_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
IppStatus ippsSynthesisFilterGetSize_SBR_RToR_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
```

Parameters

<code>pSizeSpec</code>	Address of the <code>FilterSpec_SBR</code> specification structure size value in bytes.
<code>pSizeInitBuf</code>	Address size of the buffer for initialization functions in bytes.
<code>pSizeWorkBuf</code>	Address of the work buffer size value in bytes.

Description

These functions are declared in the `ippac.h` file. The functions return sizes of the work buffer, specification structure and initialization buffer for `Init` functions for Synthesis Filter bank.

`ippsSynthesisFilterGetSize_SBR_CToR_32fc32f` returns the specified sizes for [ippsSynthesisFilter_SBR_CToR_32fc32f_D2L](#).

`ippsSynthesisFilterGetSize_SBR_CToR_32f` returns the specified sizes for [ippsSynthesisFilter_SBR_CToR_32f_D2L](#).

`ippsSynthesisFilterGetSize_SBR_RToR_32f` returns the specified sizes for [ippsSynthesisFilter_SBR_RToR_32f_D2L](#).

The functions `ippsSynthesisFilterGetSize_SBR_CToR_32f` and `ippsSynthesisFilterGetSize_SBR_RToR_32f` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

SynthesisDownFilterGetSize_SBR

Returns size of `FilterSpec_SBR` specification structures, init and work buffers.

Syntax

```
IppStatus ippsSynthesisDownFilterGetSize_SBR_CToR_32fc32f(int* pSizeSpec, int* pSizeInitBuf, int* pSizeWorkBuf);  
IppStatus ippsSynthesisDownFilterGetSize_SBR_CToR_32f(int* pSizeSpec, int* pSizeInitBuf, int* pSizeWorkBuf);  
IppStatus ippsSynthesisDownFilterGetSize_SBR_RToR_32f(int* pSizeSpec, int* pSizeInitBuf, int* pSizeWorkBuf);
```

Parameters

<code>pSizeSpec</code>	Address of the <code>FilterSpec_SBR</code> specification structure size value in bytes.
<code>pSizeInitBuf</code>	Address size of the buffer for initialization functions in bytes.
<code>pSizeWorkBuf</code>	Address of the work buffer size value in bytes.

Description

These functions are declared in the `ippac.h` file. The functions return sizes of the work buffer, specification structure and initialization buffer for `Init` functions for Synthesis Down Filter bank.

`ippsSynthesisDownFilterGetSize_SBR_CToR_32fc32f` returns the specified sizes for [ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L](#).

`ippsSynthesisDownFilterGetSize_SBR_CToR_32f` returns the specified sizes for [ippsSynthesisDownFilter_SBR_CToR_32f_D2L](#).

`ippsSynthesisDownFilterGetSize_SBR_RToR_32f` returns the specified sizes for [ippsSynthesisDownFilter_SBR_RToR_32f_D2L](#).

The functions `ippsSynthesisDownFilterGetSize_SBR_CToR_32f` and `ippsSynthesisDownFilterGetSize_SBR_RToR_32f` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

Initialization

These functions initialize the `FilterSpec` context.

AnalysisFilterInit_SBR

Initializes analysis specification structure.

Syntax

```

IppStatus ippsAnalysisFilterInit_SBR_RToC_32f32fc(IppsFilterSpec_SBR_C_32fc**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsAnalysisFilterInit_SBR_RToC_32f(IppsFilterSpec_SBR_C_32f**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsAnalysisFilterInit_SBR_RToR_32f(IppsFilterSpec_SBR_R_32f**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
    
```

Parameters

<i>ppFilterSpec</i>	Pointer to where pointer to the <code>FilterSpec_SBR</code> specification structure is written. Only the main function uses this structure.
<i>pMemSpec</i>	Pointer to the area for the <code>FilterSpec_SBR</code> specification structure.
<i>pInitBuf</i>	Pointer to the Init buffer.

Description

These functions are declared in the `ippac.h` file. The functions initialize the analysis structures:

`ippsAnalysisFilterInit_SBR_RToC_32f32fc` initializes the specification structure for [ippsAnalysisFilter_SBR_RToC_32f32fc_D2L](#).

`ippsAnalysisFilterInit_SBR_RToC_32f` initializes the specification structure for [ippsAnalysisFilter_SBR_RToC_32f_D2L](#).

`ippsAnalysisFilterInit_SBR_RToR_32f` initializes the specification structure for [ippsAnalysisFilter_SBR_RToR_32f_D2L](#).

The functions `ippsAnalysisFilterInit_SBR_RToC_32f` and `ippsAnalysisFilterInit_SBR_RToR_32f` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

SynthesisFilterInit_SBR

Initializes synthesis specification structure.

Syntax

```
IppStatus ippsSynthesisFilterInit_SBR_CToR_32fc32f(IppsFilterSpec_SBR_C_32fc**  
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

```
IppStatus ippsSynthesisFilterInit_SBR_CToR_32f(IppsFilterSpec_SBR_C_32f**  
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

```
IppStatus ippsSynthesisFilterInit_SBR_RToR_32f(IppsFilterSpec_SBR_R_32f**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

Parameters

<i>ppFilterSpec</i>	Pointer to where pointer to the <code>FilterSpec_SBR</code> specification structure is written. Only the main function uses this structure.
<i>pMemSpec</i>	Pointer to the area for the <code>FilterSpec_SBR</code> specification structure.
<i>pInitBuf</i>	Pointer to the <code>Init</code> buffer.

Description

These functions are declared in the `ippac.h` file. The functions initialize the synthesis structures:

`ippsSynthesisFilterInit_SBR_CToR_32fc32f` initializes the specification structure for [ippsSynthesisFilter_SBR_CToR_32fc32f_D2L](#).

`ippsSynthesisFilterInit_SBR_CToR_32f` initializes the specification structure for [ippsSynthesisFilter_SBR_CToR_32f_D2L](#).

`ippsSynthesisFilterInit_SBR_RToR_32f` initializes the specification structure for [ippsSynthesisFilter_SBR_RToR_32f_D2L](#).

The functions `ippsSynthesisFilterInit_SBR_CToR_32f` and `ippsSynthesisFilterInit_SBR_RToR_32f` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

SynthesisDownFilterInit_SBR

Initializes synthesis down specification structure.

Syntax

```
IppStatus
    ippsSynthesisDownFilterInit_SBR_CToR_32fc32f(IppsFilterSpec_SBR_C_32fc**
        ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsSynthesisDownFilterInit_SBR_CToR_32f(IppsFilterSpec_SBR_C_32f**
        ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsSynthesisDownFilterInit_SBR_RToR_32f(IppsFilterSpec_SBR_R_32f**
        ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

Parameters

<i>ppFilterSpec</i>	Pointer to where pointer to the FilterSpec_SBR specification structure is written. Only the main function uses this structure.
<i>pMemSpec</i>	Pointer to the area for the FilterSpec_SBR specification structure.
<i>pInitBuf</i>	Pointer to the Init buffer.

Description

These functions are declared in the `ippac.h` file. The functions initialize the down sample synthesis structures:

`ippsSynthesisDownFilterInit_SBR_CToR_32fc32f` initializes the specification structure for [ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L](#).

`ippsSynthesisDownFilterInit_SBR_CToR_32f` initializes the specification structure for [ippsSynthesisDownFilter_SBR_CToR_32f_D2L](#).

`ippsSynthesisDownFilterInit_SBR_RToR_32f` initializes the specification structure for [ippsSynthesisDownFilter_SBR_RToR_32f_D2L](#).

The functions `ippsSynthesisDownFilterInit_SBR_CToR_32f` and `ippsSynthesisDownFilterInit_SBR_RToR_32f` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

QMF Main Functions

AnalysisFilter_SBR

Splits time domain signal output from the core decoder into 32 subband signals.

Syntax

```

IppStatus ippsAnalysisFilter_SBR_RToC_32f_32fc_D2L(const Ipp32f* pSrc, Ipp32fc*
    pDst[], const Ipp32f* pSbrTableWindowDown, int NumLoop, int offset, int kx,
    const IppsFilterSpec_SBR_C_32fc* pFilterSpec, Ipp8u* pWorkBuf);

IppStatus ippsAnalysisFilter_SBR_RToC_32f_D2L(const Ipp32f* pSrc, Ipp32f*
    pDstRe[], Ipp32f* pDstIm[], const Ipp32f* pSbrTableWindowDown, int NumLoop,
    int offset, int kx, const IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u*
    pWorkBuf);

IppStatus ippsAnalysisFilter_SBR_RToR_32f_D2L(const Ipp32f* pSrc, Ipp32f*
    pDst[], const Ipp32f* pSbrTableWindowDown, int NumLoop, int offset, int kx,
    const IppsFilterSpec_SBR_R_32f* pFilterSpec, Ipp8u* pWorkBuf);

```

Parameters

<code>pSrc</code>	Pointer to the input audio signal.
<code>pDst[]</code>	Array of pointers, holds subband samples.
<code>pDstRe[]</code>	Array of pointers that contains real parts of subband samples.
<code>pDstIm[]</code>	Array of pointers that contains imaginary parts of subband samples.
<code>pSbrTableWindowDown</code>	Pointer to the window table that is used by decoder SBR Analysis Filter. See details below.

<i>offset</i>	Desired displacement in number of rows when the matrix <i>pDst</i> is calculated; <i>offset</i> ≥ 0
<i>NumLoop</i>	Parameter equal to 32 if frame size of the core decoded signal is 1024 30 if frame size of the core decoded signal is 960.
<i>kx</i>	First SBR subband in the SBR range ($0 \leq kx < 32$).
<i>pFilterSpec</i>	Pointer to the <i>FilterSpec_SBR</i> specification structure.
<i>pWorkBuf</i>	Pointer to the work buffer.

Description

These functions are declared in the `ippac.h` file. The functions filter input signal according to *pFilterSpec* specification parameters.

The function `ippsAnalysisFilter_SBR_RToC_32f32fc_D2L` uses the real input data type and complex output data type, processes the input real array *pSrc* and stores the result in the complex matrix *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function [ippsAnalysisFilterGetSize_SBR_RToC_32f32fc](#).
- This function initializes the process using function [ippsAnalysisFilterInit_SBR_RToC_32f32fc](#). Deallocate *pInitBuf* after completing initialization.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32fc*` type.

The function `ippsAnalysisFilter_SBR_RToC_32f_D2L` uses the real data type and stores the result separately in *pDstRe* matrix and *pDstIm* matrix.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function [ippsAnalysisFilterGetSize_SBR_RToC_32f](#).
- This function initializes the process using function [ippsAnalysisFilterInit_SBR_RToC_32f](#). After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32f*` type.

The functionality of `ippsAnalysisFilter_SBR_RToC_32f32fc_D2L` and `ippsAnalysisFilter_SBR_RToC_32f_D2L` is described as follows:

Step 1. Buffer Updating

$$x[n] = x[n - 32], \quad 320 > n \geq 32$$

$$x[n] = pSrc[1 \times 32 + 31 - n], \quad 31 \geq n \geq 0$$

Step 2. Polyphase Filtering

$$u[n] = \sum_{j=0}^4 x[n + j \times 64] \times pSbrTableWindowDown[n + j \times 64], \quad 0 \leq n < 64$$

Step 3. Special Fourier Transform

$$pDst[l + offset][k] = 2 \sum_{n=0}^{63} u[n] \times \exp\left[i \frac{\pi}{64} \times \left(k + \frac{1}{2}\right) \left(2n - \frac{1}{2}\right)\right], \quad 0 \leq k < 32$$

Step 4. Clearing Upper Part of Spectrum

$$pDst[l + offset][k] = 0, \quad k \times 32 < k < 32.$$

Note that Steps 1 through 4 are repeated for $0 \leq l < NumLoop$ and

$$pSbrTableWindowDown[n] = c[2 \times n], \quad n = 0, \dots, 320,$$

where coefficients $c[i]$ are coefficients of the QMF bank window from [\[ISO14496\]](#).

The function `ippsAnalysisFilter_SBR_RToR_32f_D2L` uses the real data type, processes the input real array `pSrc` and stores the result in the real matrix `pDst`.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function [ippsAnalysisFilterGetSize_SBR_RToR_32f](#).
- This function initializes the process using function [ippsAnalysisFilterInit_SBR_RToR_32f](#). After completing initialization `pInitBuf` can be deallocated.
- `pFilterSpec` specification parameter has to be of `IppsFilterSpec_SBR_R_32f*` type.

The functionality of `ippsAnalysisFilter_SBR_RToR_32f_D2L` can be described in the same way as the above `AnalysisFilter_SBR_D2L` functions but Step 3 changes as follows:

Step 3. Special Fourier Transform

$$pDst[l + offset][k] = 2 \sum_{n=0}^{63} u[n] \times \cos \left[\frac{\pi}{64} \times \left(k + \frac{1}{2} \right) (2n - 96) \right], \quad 0 \leq k < 32 .$$

The functions `ippsAnalysisFilter_SBR_RToC_32f_D2L` and `ippsAnalysisFilter_SBR_RToR_32f_D2L` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the identifier <code>pFilterSpec</code> is incorrect.

SynthesisFilter_SBR

Transforms SBR-processed subband signals into time domain samples.

Syntax

```
IppStatus ippsSynthesisFilter_SBR_CToR_32fc32f_D2L (const Ipp32fc* pSrc[],
    Ipp32f* pDst, const Ipp32f* pSbrTableWindow, int NumLoop, const
    IppsFilterSpec_SBR_C_32fc* pFilterSpec, Ipp8u* pWorkBuf);

IppStatus ippsSynthesisFilter_SBR_CToR_32f_D2L (const Ipp32f* pSrcRe[], const
    Ipp32f* pSrcIm[], Ipp32f* pDst, const Ipp32f* pSbrTableWindow, int NumLoop,
    const IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u* pWorkBuf);

IppStatus ippsSynthesisFilter_SBR_RToR_32f_D2L (const Ipp32f* pSrc[], Ipp32f*
    pDst, const Ipp32f* pSbrTableWindow, int NumLoop, const
    IppsFilterSpec_SBR_R_32f* pFilterSpec, Ipp8u* pWorkBuf);
```

Parameters

<code>pSrc[]</code>	Array of pointers, holds SBR-processed subband signals.
---------------------	---

<i>pSrcRe[]</i>	Array of pointers that contains real parts of SBR-processed subband signals.
<i>pSrcIm[]</i>	Array of pointers that contains imaginary parts of SBR-processed subband signals.
<i>pDst</i>	Pointer to the output vector, holds time domain output samples.
<i>pSbrTableWindow</i>	Pointer to the window table that is used by SBR Analysis Filter. See details below.
<i>NumLoop</i>	Parameter equal to 32 if frame size of the core decoded signal is 1024 30 if frame size of the core decoded signal is 960.
<i>pFilterSpec</i>	Pointer to the <code>FilterSpec_SBR</code> specification structure.
<i>pWorkBuf</i>	Pointer to the work buffer used by the filter.

Description

These functions are declared in the `ippac.h` file. The functions transform SBR-processed subband signals into time domain samples according to *pFilterSpec* specification parameters.

The function `ippsSynthesisFilter_SBR_CToR_32fc32f_D2L` uses the complex input data type and real output data type, processes the input complex array of pointers *pSrc* and stores the result in the real vector *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function [ippsSynthesisFilterGetSize_SBR_CToR_32fc32f](#).
- This function initializes the process using function [ippsSynthesisFilterInit_SBR_CToR_32fc32f](#). Deallocate *pInitBuf* after completing initialization.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32fc*` type.

The function `ippsSynthesisFilter_SBR_CToR_32f_D2L` uses the real data type and stores the result separately in *pSrcRe* matrix and *pSrcIm* matrix.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function [ippsSynthesisFilterGetSize_SBR_CToR_32f](#).
- This function initializes the process using function [ippsSynthesisFilterInit_SBR_CToR_32f](#). After completing initialization *pInitBuf* can be deallocated.

- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32f*` type.

The functionality of `ippsSynthesisFilter_SBR_CToR_32fc32f_D2L` and `ippsSynthesisFilter_SBR_CToR_32f_D2L` is described as follows:

Step 1. Buffer Updating

$$v[n] = v[n - 128], \quad 1280 > n \geq 128$$

Step 2. Special Fourier Transform

$$v[n] = \frac{1}{64} \sum_{k=0}^{63} \operatorname{Re} \left\{ pSrc[l][k] \times \exp \left[i \frac{\pi}{128} \times \left(k + \frac{1}{2} \right) (2n - 255) \right] \right\}, \quad 0 \leq n < 128$$

Step 3. Polyphase Filtering

$$\begin{aligned} v[128 \times n + k] &= v[256 \times n + k] \times pSbrTableWindow[128 \times n + k] \\ v[128 \times n + 64 + k] &= v[256 \times n + 192 + k] \times pSbrTableWindow[128 \times n + 64 + k] \end{aligned} \quad , \quad \begin{cases} 0 \leq n < 5 \\ 0 \leq k < 64 \end{cases}$$

Step 4. Output Vector Updating

$$pDst[64 \times l + k] = \sum_{n=0}^9 w[64 \times n + k], \quad 0 \leq k < 64.$$

Note that Steps 1 through 4 are repeated for $0 \leq l < \text{NumLoop}$ and $pSbrTableWindow[n] = c[n], n = 0, \dots, 640$,

where coefficients $c[i]$ are coefficients of the QMF bank window from [\[ISO14496\]](#).

The function `ippsSynthesisFilter_SBR_RToR_32f_D2L` uses the real data type, processes the input real matrix *pSrc* and stores the result in the real array *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function [ippsSynthesisFilterGetSize_SBR_RToR_32f](#).
- This function initializes the process using function [ippsSynthesisFilterInit_SBR_RToR_32f](#). After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_R_32f*` type.

The functionality of `ippsSynthesisFilter_SBR_RToR_32f` can be described in the same way as the above `SynthesisFilter_SBR` functions but Step 2 changes as follows:

Step 2. Special Fourier Transform

$$v[n] = \frac{1}{32} \sum_{k=0}^{63} pSrc[l][k] \times \cos\left[\frac{\pi}{128} \times \left(k + \frac{1}{2}\right)(2n - 64)\right], \quad 0 \leq n < 128.$$

The functions `ippsSynthesisFilter_SBR_CToR_32f_D2L` and `ippsSynthesisFilter_SBR_RToR_32f_D2L` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the identifier <code>pFilterSpec</code> is incorrect.

SynthesisDownFilter_SBR

Transforms SBR-processed subband signals into time domain samples and performs downsampling at the same time.

Syntax

```

IppStatus ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L (const Ipp32fc* pSrc[],
    Ipp32f* pDst, const Ipp32f* pSbrTableWindowDown, const int NumLoop, const
    IppsFilterSpec_SBR_C_32fc* pFilterSpec, Ipp8u* pWorkBuf);

IppStatus ippsSynthesisDownFilter_SBR_CToR_32f_D2L(const Ipp32f* pSrcRe[],
    const Ipp32f* pSrcIm[], Ipp32f* pDst, const Ipp32f* pSbrTableWindow, const
    int NumLoop, const IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u* pWorkBuf);

IppStatus ippsSynthesisDownFilter_SBR_RToR_32f_D2L(const Ipp32f* pSrc[],
    Ipp32f* pDst, const Ipp32f* pSbrTableWindow, const int NumLoop, const
    IppsFilterSpec_SBR_R_32f* pFilterSpec, Ipp8u* pWorkBuf);

```

Parameters

<i>pSrc</i> []	Array of pointers, holds SBR-processed subband signals.
<i>pSrcRe</i> []	Array of pointers that contains real parts of SBR-processed subband signals.
<i>pSrcIm</i> []	Array of pointers that contains imaginary parts of SBR-processed subband signals.
<i>pDst</i>	Pointer to the output vector, holds time domain output samples.
<i>pSbrTableWindowDown</i>	Pointer to the window table that is used by SBR Analysis Filter. See details below.
<i>NumLoop</i>	Parameter equal to 32 if frame size of the core decoded signal is 1024 30 if frame size of the core decoded signal is 960.
<i>pFilterSpec</i>	Pointer to the <code>FilterSpec_SBR</code> specification structure.
<i>pWorkBuf</i>	Pointer to the work buffer used by the filter.

Description

These functions are declared in the `ippac.h` file. The functions transform SBR-processed subband signals into time domain samples and perform downsampling at the same time according to *pFilterSpec* specification parameters.

The function `ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L` uses the complex input data type and real output data type, processes the input complex matrix *pSrc* and stores the result in the real array *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function [ippsSynthesisDownFilterGetSize_SBR_CToR_32fc32f](#).
- This function initializes the process using function [ippsSynthesisDownFilterInit_SBR_CToR_32fc32f](#). Deallocate *pInitBuf* after completing initialization.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32fc*` type.

The function `ippsSynthesisDownFilter_SBR_CToR_32f_D2L` uses the real data type and stores the result separately in *pSrcRe* matrix and *pSrcIm* matrix.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function [ippsSynthesisDownFilterGetSize_SBR_CToR_32f](#).

- This function initializes the process using function [ippsSynthesisDownFilterInit_SBR_CToR_32f](#). After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32f*` type.

The functionality of `ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L` and `ippsSynthesisDownFilter_SBR_CToR_32f_D2L` is described as follows:

Step 1. Buffer Updating

$$v[n] = v[n - 64], \quad 64 > n \geq 64$$

Step 2. Special Fourier Transform

$$v[n] = \frac{1}{64} \sum_{k=0}^{31} \operatorname{Re} \left\{ pSrc[l][k] \times \exp \left[i \frac{\pi}{64} \times \left(k + \frac{1}{2} \right) (2n - 127) \right] \right\}, \quad 0 \leq n < 64$$

Step 3. Polyphase Filtering

$$\begin{aligned} v[64 \times n + k] &= v[128 \times n + k] \times pSbrTableWindow[128 \times n + k] \\ v[64 \times n + 32 + k] &= v[128 \times n + 96 + k] \times pSbrTableWindow[128 \times n + 96 + k] \end{aligned}, \quad \begin{cases} 0 \leq n < 5 \\ 0 \leq k < 32 \end{cases}$$

Step 4. Output Vector Updating

$$pDst[32 \times l + k] = \sum_{n=0}^9 w[32 \times n + k], \quad 0 \leq k < 32.$$

Note that Steps 1 through 4 are repeated for $0 \leq l < \text{NumLoop}$ and

$$pSbrTableWindowDown[n] = c[2 \times n], \quad n = 0, \dots, 320,$$

where coefficients $c[i]$ are coefficients of the QMF bank window from [\[ISO14496\]](#).

The function `ippsSynthesisDownFilter_SBR_RToR_32f_D2L` uses the real data type, processes the input real matrix *pSrc* and stores the result in the real array *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function [ippsSynthesisDownFilterGetSize_SBR_RToR_32f](#).
- This function initializes the process using function [ippsSynthesisDownFilterInit_SBR_RToR_32f](#). After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_R_32f*` type.

The functionality of `ippsSynthesisDownFilter_SBR_RToR_32f_D2L` can be described in the same way as the above `SynthesisFilter_SBR` functions but Step 2 changes as follows:

Step 2. Special Fourier Transform

$$v[n] = \frac{1}{32} \sum_{k=0}^{31} pSrc[1][k] \times \cos\left[\frac{\pi}{64} \times \left(k + \frac{1}{2}\right)(2n - 32)\right], \quad 0 \leq n < 64.$$

The functions `ippsSynthesisDownFilter_SBR_CToR_32f_D2L` and `ippsSynthesisDownFilter_SBR_RToR_32f_D2L` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the identifier <code>pFilterSpec</code> is incorrect.

Prediction

SBR prediction functions provide inverse filtering for correct reconstruction of high frequency transmitted signals.

PredictCoef_SBR

Obtains prediction filter coefficients using covariance method.

Syntax

```
IppStatus ippsPredictCoef_SBR_C_32fc_D2L(const Ipp32fc* pSrc[], Ipp32fc*
    pAlpha0, Ipp32fc* pAlpha1, int k0, int len);

IppStatus ippsPredictCoef_SBR_C_32f_D2L(const Ipp32f* pSrcRe[], const Ipp32f*
    pSrcIm[], Ipp32f* pAlpha0Re, Ipp32f* pAlpha0Im, Ipp32f* pAlpha1Re, Ipp32f*
    pAlpha1Im, int k0, int len);
```

```
IppStatus ippSPredictCoef_SBR_R_32f_D2L(const Ipp32f* pSrc[], Ipp32f* pAlpha0,
    Ipp32f* pAlpha1, int k0, int len);
```

Parameters

<i>pSrc[]</i>	Array of pointers that holds input real or complex matrix ([40][32]), holds Low Frequency QMF-processed subband signals.
<i>pSrcRe[]</i>	Array of pointers ([40][32]) that contains real parts of Low Frequency SBR-processed subband signals.
<i>pSrcIm[]</i>	Array of pointers ([40][32]) that contains imaginary parts of Low Frequency SBR-processed subband signals.
<i>pAlpha0, pAlpha1</i>	Prediction filter coefficients used by High Frequency filter.
<i>pAlpha0Re, pAlpha1Re</i>	Pointers to the arrays that contain real parts of the corresponding complex filter coefficients.
<i>pAlpha0Im, pAlpha1Im</i>	Pointers to the arrays that contain imaginary parts of the corresponding complex filter coefficients.
<i>k0</i>	First QMF subband in the <code>f_master</code> table.
<i>len</i>	Size autocorrelation.

Description

These functions are declared in the `ippac.h` file. The function `PredictCoef_SBR` obtains prediction filter coefficients using covariance method.

The function `ippSPredictCoef_SBR_C_32fc` uses the complex input data type and complex output data type, processes the input complex matrix *pSrc* and stores the result in complex arrays *pAlpha0* and *pAlpha1*.

The function `ippSPredictCoef_SBR_C_32f_D2L` uses the real input and output data type, uses complex input data separately in matrix *pSrcRe* and matrix *pSrcIm* and stores complex output data separately in *pAlpha0Re*, *pAlpha0Im*, *pAlpha1Re*, *pAlpha1Im*.

The functions `ippSPredictCoef_SBR_C_32fc_D2L` and `ippSPredictCoef_SBR_C_32f_D2L` are used in high quality SBR mode.

The function `ippSPredictCoef_SBR_R_32f_D2L` uses the real input and output data type. This function is used in low power SBR mode.

The prediction filter coefficients are obtained with the covariance method using the following formula:

$$\phi_k(i, j) = \sum_{n=0}^{len-1} pSrc[n-i+2][k] \cdot conj(pSrc[n-j+2][k]),$$

$$0 \leq i < 3, 1 \leq j < 3, 0 \leq k < k_0.$$

The coefficients $pAlpha0[k]$ and $pAlpha1[k]$ are calculated as follows:

$$l(k) = \phi_k(2, 2) \cdot \phi_k(1, 1) - \frac{1}{1 + \epsilon_{inv}} |\phi_k(1, 2)|,$$

$$Alpha1[k] = \begin{cases} \phi_k(0, 1) \times \phi_k(1, 2) - \phi_k(0, 2) \times \phi_k(1, 1), & d(k) \neq 0 \\ 0, & d(k) = 0 \end{cases}$$

$$Alpha0[k] = \begin{cases} \phi_k(0, 1) \times pAlpha1[k] \times conj(\phi_k(1, 2)), & d(k) \neq 0 \\ 0, & d(k) = 0 \end{cases}$$

The functions `ippsPredictCoef_SBR_C_32f_D2L` and `ippsPredictCoef_SBR_R_32f_D2L` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

String Functions

11

This chapter describes Intel[®] IPP functions that perform operations with a text. First part describes the functions for simple string manipulation. Second part contains functions that perform more sophisticated matching operation using patterns of the regular expressions.

The full list of functions in this group is given in [Table 11-1](#).

Table 11-1 Intel IPP String Functions

Function Base Name	Operation
String Manipulation Functions	
Find, FindRev	Looks for the first occurrence of the substring matching the specified string.
FindC, FindRevC	Looks for the first occurrence of the specified element within the source string.
FindCAny, FindRevCAny	Looks for the first occurrence of any element of the specified array within the source string.
Insert	Inserts a string into another string.
Remove	Removes a specified number of elements from the string.
Compare	Compares two strings of the fixed length.
CCompareIgnoreCase, CompareIgnoreCaseLatin	Compares two strings of the fixed length ignoring case.
Equal	Compares two strings of the fixed length for equality.
TrimC	Deletes all occurrences of a specified symbol both in the beginning and in the end of the string.
TrimCAny	Deletes all occurrences of any of the specified symbols both in the beginning and in the end of the source string.

Table 11-1 Intel IPP String Functions

Function Base Name	Operation
TrimStartCAny TrimEndCAny	Deletes all occurrences of any of the specified symbols either in the beginning or in the end of the source string, respectively.
ReplaceC	Replaces all occurrences of a specified element in the source string with another element.
Uppercase, UppercaseLatin	Converts alphabetic characters of a string to all uppercase symbols.
Lowercase, LowercaseLatin	Converts alphabetic characters of a string to all lowercase symbols.
Hash	Calculates a hash value for the string.
Concat	Concatenates several strings together.
ConcatC	Concatenates several strings together and inserts symbol delimiters between them.
SplitC	Splits source string into separate parts.
Functions for Work with Regular Expressions	
RegExpInitAlloc	Allocates memory and initializes the structure for processing matching operation with regular expressions..
RegExpFree	Frees the memory allocated for a regular expression state structure..
RegExpInit	Initializes the structure for processing matching operation with regular expressions.
RegExpGetSize	Computes the size of the regular expression state structure..
RegExpSetMatchLimit	Sets the value of the <i>matchLimit</i> parameter.
RegExpFind	Looks for the occurrences of the substrings matching the specified regular expression.

String Manipulation

This section describes the Intel IPP functions that perform operations with strings. Intel IPP string functions do not consider zero as the end of the string, but require that the length of the string (number of elements) be specified explicitly.

Overlapping of the strings is not supported (for not in-place operations).

Intel IPP string functions operate with two data types, `Ipp8u` and `Ipp16u`.

Find, FindRev

Looks for the first occurrence of the substring matching the specified string.

Syntax

```
IppStatus ippsFind_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind, int  
    lenFind, int* pIndex);  
IppStatus ippsFind_16u(const Ipp16u* pSrc, int len, const Ipp16u* pFind, int  
    lenFind, int* pIndex);  
IppStatus ippsFindRev_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind, int  
    lenFind, int* pIndex);  
IppStatus ippsFindRev_16u(const Ipp16u* pSrc, int len, const Ipp16u* pFind, int  
    lenFind, int* pIndex);
```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>len</i>	Number of elements in the source string.
<i>pFind</i>	Pointer to the reference string.
<i>lenFind</i>	Number of elements in the reference string.
<i>pIndex</i>	Pointer to the result index.

Description

The functions `ippsFind` and `ippsFindRev` are declared in the `ippch.h` file. These functions search through the source string *pSrc* for a substring of elements that match the specified reference string *pFind*. Starting point of the first occurrence of the matching substring is stored in *pIndex*. If no matching substring is found, then *pIndex* is set to -1. The function `ippsFindRev` searches the source string in the reverse direction. The search is case-sensitive.

[Example 11-1](#) shows how to use the function `ippsFind_8u`.

Example 11-1 Using the function `ippsFind`

```
IppStatus find( void ) {
    IppStatus st;
    int result;
    char str[] = "matching";
    char string[] = "This function looks for the matching substring";
    char fmt1[] = "          1          2          3          4          5";
    char fmt2[] = "012345678901234567890123456789012345678901234567890";
    printf( "String to be searched:\n\t%s\n", string );
    printf( "\t%s\n\t%s\n\n", fmt1, fmt2 );
    st = ippsFind_8u((Ipp8u*)string, strlen(string), (Ipp8u*)str,
    strlen(str), &result);
    printf( "%s found at position %d\n\n", str, result );
    return st;
}
```

Output:

String to be searched:

```
This function looks for the matching substring
          1          2          3          4          5
012345678901234567890123456789012345678901234567890
```

matching found at position 28

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippsStsLengthErr</code>	Indicates an error condition if <code>len</code> or <code>lenFind</code> is negative.

FindC, FindRevC

Looks for the first occurrence of the specified element within the source string.

Syntax

```
IppStatus ippsFindC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind, int* pIndex);  
IppStatus ippsFindC_16u(const Ipp16u* pSrc, int len, Ipp16u valFind, int*  
    pIndex);  
IppStatus ippsFindRevC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind, int*  
    pIndex);  
IppStatus ippsFindRevC_16u(const Ipp16u* pSrc, int len, Ipp16u valFind, int*  
    pIndex);
```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>len</i>	Number of elements in the source string.
<i>valFind</i>	Value of the specified element.
<i>pIndex</i>	Pointer to the result index.

Description

The functions `ippsFindC` and `ippsFindRevC` are declared in the `ippch.h` file. These functions search through the source string *pSrc* for the first occurrence of the specified element with the value *valFind*. The position of this element is stored in *pIndex*. If no matching element is found, then *pIndex* is set to -1. The function `ippsFindRev` searches the source string in the reverse direction. The search is case-sensitive.

[Example 11-2](#) shows how to use the function `ippsFindC_8u`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

Example 11-2 Using the Function `ippsFindC`

```

IppStatus findc( void ) {
    IppStatus st;
    int result;
    char ch = 'r';
    char string[] = "This function looks for the specified character";
    char fmt1[] = "          1          2          3          4          5";
    char fmt2[] = "012345678901234567890123456789012345678901234567890";
    printf( "String to be searched:\n\t%s\n", string );
    printf( "\t%s\n\t%s\n\n", fmt1, fmt2 );
    /* Search forward. */
    st = ippsFindC_8u((Ipp8u*)string, strlen(string), ch, &result);
    printf( "Result:\tfirst %c found at position %d\n\n", ch, result );
    /* Search backward. */
    st = ippsFindRevC_8u((Ipp8u*)string, strlen(string), ch, &result);
    printf( "Result:\tlast %c found at position %d\n\n", ch, result );
    return st;
}

```

Output:

String to be searched:

```

    This function looks for the specified character
           1           2           3           4           5
012345678901234567890123456789012345678901234567890

```

Result: first r found at position 22

Result: last r found at position 46

FindCAny, FindRevCAny

Looks for the first occurrence of any element of the specified array within the source string.

Syntax

```
IppStatus ippsFindCAny_8u(const Ipp8u* pSrc, int len, const Ipp8u* pAnyOf, int  
    lenAnyOf, int* pIndex);  
IppStatus ippsFindCAny_16u(const Ipp16u* pSrc, int len, const Ipp16u* pAnyOf,  
    int lenAnyOf, int* pIndex);  
IppStatus ippsFindRevCAny_8u(const Ipp8u* pSrc, int len,  
    const Ipp8u* pAnyOf, int lenAnyOf, int* pIndex);  
IppStatus ippsFindRevCAny_16u(const Ipp16u* pSrc, int len, const Ipp16u*  
    pAnyOf, int lenAnyOf, int* pIndex);
```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>len</i>	Number of elements in the source string.
<i>pAnyOf</i>	Pointer to the array containing reference elements.
<i>lenAnyOf</i>	Number of elements in the array.
<i>pIndex</i>	Pointer to the result index.

Description

The functions `ippsFindCAny` and `ippsFindRevCAny` are declared in the `ippch.h` file. These functions search through the source string *pSrc* for the first occurrence of any reference element from the specified array *pAnyOf*. The position of this element is stored in *pIndex*. If no matching element is found, then *pIndex* is set to -1. The function `ippsFindRevCAny` searches the source string in the reverse direction.

The search is case-sensitive.

[Example 11-3](#) shows how to use the function `ippsFindCAny_8u`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error condition if at least one of the specified pointers is NULL.

`ippStsLengthErr` Indicates an error condition if *len* or *lenAnyOf* is negative.

Example 11-3 Using the Function `ippsFindCAny`

```
IppStatus findcany( void ) {
    IppStatus st;
    char string[] = "xyzabc";
    char AnyOf[] = "bac";
    int pos;
    st = ippsFindCAny_8u((Ipp8u*)string, strlen(string), (Ipp8u*)AnyOf,
    strlen(AnyOf), &pos);
    printf( "First a, b or c in %s is at position %d\n", string, pos );
    return st;
}
```

Output:

First a, b or c in xyzabc is at position 3

Insert

Inserts a string into another string.

Syntax

```
IppStatus ippsInsert_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pInsert,
    int insertLen, Ipp8u* pDst, int startIndex);
IppStatus ippsInsert_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u* pInsert,
    int insertLen, Ipp16u* pDst, int startIndex);
IppStatus ippsInsert_8u_I(const Ipp8u* pInsert, int insertLen, Ipp8u* pSrcDst,
    int* pSrcDstLen, int startIndex);
IppStatus ippsInsert_16u_I(const Ipp16u* pInsert, int insertLen, Ipp16u*
    pSrcDst, int* pSrcDstLen, int startIndex);
```

Parameters

pSrc Pointer to the source string.

<i>srcLen</i>	Number of elements in the source string.
<i>pInsert</i>	Pointer to the string to be inserted.
<i>insertLen</i>	Number of elements in the string to be inserted.
<i>pDst</i>	Pointer to the destination string.
<i>pSrcDst</i>	Pointer to the source and destination string for in-place operation.
<i>pSrcDstLen</i>	Pointer to the number of elements in the source and destination string for in-place operation.
<i>startIndex</i>	Index of the insertion point.

Description

The function `ippsInsert` is declared in the `ippch.h` file. This function inserts the string *pInsert* containing *insertLen* elements into a source string *pSrc* of length *srcLen*. Insertion position is specified by *startIndex*. The result is stored in the *pDst*.

The in-place flavors of `ippsInsert` insert the string *pInsert* in the source string *pSrcDst* and store the result in the destination string *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if one of the <i>srcLen</i> , <i>insertLen</i> , <i>pSrcDstLen</i> , <i>startIndex</i> is negative, or <i>startIndex</i> is greater than <i>srcLen</i> or <i>pSrcDstLen</i> .

Remove

Removes a specified number of elements from the string.

Syntax

```
IppStatus ippsRemove_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int  
    startIndex, int len);
```

```
IppStatus ippRemove_16u(const Ipp16u* pSrc, int srcLen, Ipp16u* pDst, int
    startIndex, int len);
IppStatus ippRemove_8u_I(Ipp8u* pSrcDst, int* pSrcDstLen, int startIndex, int
    len);
IppStatus ippRemove_16u_I(Ipp16u* pSrcDst, int* pSrcDstLen, int startIndex,
    int len);
```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>srcLen</i>	Number of elements in the source string.
<i>pDst</i>	Pointer to the destination string.
<i>pSrcDst</i>	Pointer to the source and destination string for in-place operation.
<i>pSrcDstLen</i>	Pointer to the number of elements in the source and destination string for in-place operation.
<i>startIndex</i>	Index of the starting point.
<i>len</i>	Number of elements to be removed.

Description

The function `ippRemove` is declared in the `ippch.h` file. This function removes the *len* elements from a source string *pSrc* of length *srcLen*. Starting position is specified by *startIndex*. The result is stored in the *pDst*.

The in-place flavors of `ippRemove` remove the *len* elements from the source string *pSrcDst* and store the result in the destination string *pSrcDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if one of the <i>srcLen</i> , <i>len</i> , <i>pSrcDstLen</i> , <i>startIndex</i> is negative, or (<i>startIndex+len</i>) is greater than <i>srcLen</i> or <i>pSrcDstLen</i> .

Compare

Compares two strings of the fixed length.

Syntax

```
IppStatus ippsCompare_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, int len,
    int* pResult);
IppStatus ippsCompare_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, int len,
    int* pResult);
```

Parameters

<i>pSrc1</i>	Pointer to the first source string.
<i>pSrc2</i>	Pointer to the second source string.
<i>len</i>	Maximum number of elements to be compared.
<i>pResult</i>	Pointer to the result.

Description

The function `ippsCompare` is declared in the `ippch.h` file. This function compares first *len* elements of two strings *pSrc1* and *pSrc2*. The value *pResult* = *pSrc1*[*i*]-*pSrc2*[*i*] is computed successively for each *i*-th element, *i*=0,...*len*-1. When the first pair of non-matching elements occurs (that is, when *pResult* is not equal to zero), the function stops operation and returns the value *pResult*. The returned value is positive when *pSrc1*[*i*]>*pSrc2*[*i*] and negative when *pSrc1*[*i*]<*pSrc2*[*i*]. If the strings are equal, the function returns *pResult* = 0. The comparison is case-sensitive.

[Example 11-4](#) shows how to use the function `ippsCompare_8u`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

Example 11-4 Using the Functions `ippsCompare` and `ippsCompareIgnoreCaseLatin`

```

IppStatus compare( void ) {
    IppStatus st;
    char tmp[20];
    int result;
    char string1[] = "These functions compare two strings";
    char string2[] = "These FUNCTIONS compare two strings";
    /* Case sensitive */
    printf( "Compare strings:\n\t%s\n\t%s\n\n", string1, string2 );
    st = ippsCompare_8u((Ipp8u*)string1, (Ipp8u*)string2, strlen(string1),
    &result);
    if( result > 0 )
        strcpy( tmp, "greater than" );
    else if( result < 0 )
        strcpy( tmp, "less than" );
    else
        strcpy( tmp, "equal to" );
    printf( "\tippsCompare_8u:                               String 1 is %s string 2\n",
    tmp );
    /* Case insensitive */
    st = ippsCompareIgnoreCaseLatin_8u((Ipp8u*)string1, (Ipp8u*)string2,
    strlen(string1), &result);
    if( result > 0 )
        strcpy( tmp, "greater than" );
    else if( result < 0 )
        strcpy( tmp, "less than" );
    else
        strcpy( tmp, "equal to" );
    printf( "\tippsCompareIgnoreCaseLatin_8u: String 1 is %s string 2\n",
    tmp );
    return st;
}

```

Output:

Compare strings:

```

    These functions compare two strings
    These FUNCTIONS compare two strings

```

```

    ippsCompare_8u:                               String 1 is greater than string 2
    ippsCompareIgnoreCaseLatin_8u: String 1 is equal to string 2

```

CompareIgnoreCase, CompareIgnoreCaseLatin

Compares two strings of the fixed length ignoring case.

Syntax

```
IppStatus ippsCompareIgnoreCase_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,  
    int len, int* pResult);  
IppStatus ippsCompareIgnoreCaseLatin_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,  
    int len, int* pResult);  
IppStatus ippsCompareIgnoreCaseLatin_16u(const Ipp16u* pSrc1, const Ipp16u*  
    pSrc2, int len, int* pResult);
```

Parameters

<i>pSrc1</i>	Pointer to the first source string.
<i>pSrc2</i>	Pointer to the second source string.
<i>len</i>	Maximum number of elements to be compared.
<i>pResult</i>	Pointer to the result.

Description

The functions `ippsCompareIgnoreCase` and `ippsCompareIgnoreCaseLatin` are declared in the `ippch.h` file. These functions compare first *len* elements of two strings *pSrc1* and *pSrc2*. If all pairs of elements in the strings are equal, the function returns *pResult* = 0.

If the pair of non-matching elements occurs in the *i*-th position, the function stops operation and returns *pResult*. The returned value is positive when *pSrc1*[*i*] > *pSrc2*[*i*] and negative when *pSrc1*[*i*] < *pSrc2*[*i*]. The comparison is case-insensitive.

The function `ippsCompareIgnore` operates with Unicode characters.

The function `ippsCompareIgnoreLatin` operates with ASCII characters.

[Example 11-4](#) shows how to use the function `ippsCompareIgnoreCaseLatin_8u`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

Equal

Compares two string of the fixed length for equality.

Syntax

```
IppStatus ippsequal_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, int len,
    int* pResult);
IppStatus ippsequal_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, int len,
    int* pResult);
```

Parameters

<i>pSrc1</i>	Pointer to the first source string.
<i>pSrc2</i>	Pointer to the second source string.
<i>len</i>	Maximum number of elements to be compared.
<i>pResult</i>	Pointer to the result.

Description

The function `ippsequal` is declared in the `ippch.h` file. This function compares first *len* elements of two strings *pSrc1* and *pSrc2*. Each element of the first string is compared with the corresponding element of the second string. When the first pair of non-matching elements is found, the function stops operation and stores 0 in *pResult*. If the strings are equal, the function stores 1 in *pResult*.

The comparison is case-sensitive.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

TrimC

Deletes all occurrences of a specified symbol in the beginning and in the end of the string.

Syntax

```
IppStatus ippsTrimC_8u(const Ipp8u* pSrc, int srcLen, Ipp8u odd, Ipp8u* pDst,
    int* pDstLen);
IppStatus ippsTrimC_16u(const Ipp16u* pSrc, int srcLen, Ipp16u odd, Ipp16u*
    pDst, int* pDstLen);
IppStatus ippsTrimC_8u_I(Ipp8u* pSrcDst, int* pLen, Ipp8u odd);
IppStatus ippsTrimC_16u_I(Ipp16u* pSrcDst, int* pLen, Ipp16u odd);
```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>srcLen</i>	Number of elements in the source string.
<i>pSrcDst</i>	Pointer to the source and destination string for the in-place operation.
<i>pDst</i>	Pointer to the destination string.
<i>pDstLen</i>	Pointer to the computed number of elements in the destination string.
<i>pLen</i>	Pointer to the number of elements in the source and destination string for the in-place operation.
<i>odd</i>	Symbol to be deleted.

Description

The function `ippsTrimC` is declared in the `ippch.h` file. This function deletes all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrc* containing *srcLen* elements. The function stores the result string containing *pDstLen* elements in *pDst*.

The in-place flavors of `ippsTrimC` delete all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrcDst* containing *pLen* elements. These functions store the result string containing *pLen* elements in *pSrcDst*.

The operation is case-sensitive.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>srcLen</i> or <i>pLen</i> is negative.

TrimCAny TrimStartCAny TrimEndCAny

Deletes all occurrences of any of the specified symbols in the beginning and in the end of the source string.

Syntax

```

IppStatus ippTrimCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pTrim,
    int trimLen, Ipp8u* pDst, int* pDstLen);

IppStatus ippTrimCAny_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u* pTrim,
    int trimLen, Ipp16u* pDst, int* pDstLen);

IppStatus ippTrimStartCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u*
    pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);

IppStatus ippTrimStartCAny_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u*
    pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);

IppStatus ippTrimEndCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pTrim,
    int trimLen, Ipp8u* pDst, int* pDstLen);

IppStatus ippTrimEndCAny_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u*
    pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);

```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>srcLen</i>	Number of elements in the source string.
<i>pTrim</i>	Pointer to the array containing the specified elements.
<i>trimLen</i>	Number of elements in the array.
<i>pDst</i>	Pointer to the destination string.

pDstLen Pointer to the computed number of elements in the destination string.

Description

The functions `ippsTrimCAny`, `ippsTrimStartCAny`, and `ippsTrimEndCAny` are declared in the `ippch.h` file.

The function `ippsTrimCAny` deletes all occurrences of any of the specified elements stored in the array *pTrim* if they are present either in the beginning or in the end of the source string *pSrc*, and stores the result string containing *pDstLen* elements in *pDst*. The function stops operation when it finds the first non-matching element.

The functions `ippsTrimStartCAny` and `ippsTrimEndCAny` perform this operation only in the beginning or in the end of the source string *pSrc*, respectively.

The operation is case-sensitive.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>srcLen</i> or <i>trimLen</i> is negative.

ReplaceC

Replaces all occurrences of a specified element in the source string with another element.

Syntax

```
IppStatus ippsReplaceC_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, Ipp8u  
    oldVal, Ipp8u newVal);  
  
IppStatus ippsReplaceC_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len, Ipp16u  
    oldVal, Ipp16u newVal);
```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>len</i>	Number of elements in the source string.
<i>pDst</i>	Pointer to the destination string.

<i>oldVal</i>	Element to be replaced.
<i>newVal</i>	Element that replaces <i>oldVal</i> .

Description

The function `ippsReplaceC` is declared in the `ippch.h` file. This function replaces all occurrences of a specified element *oldVal* in the source string *pSrc* with another specified element *newVal*, and stores the new string in the *pDst*. The operation is case-sensitive.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

Uppercase, UppercaseLatin

Converts alphabetic characters of a string to all uppercase symbols.

Syntax

```

IppStatus ippsUppercase_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsUppercase_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsUppercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsUppercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsUppercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsUppercaseLatin_16u_I(Ipp16u* pSrcDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>pDst</i>	Pointer to the destination string.
<i>pSrcDst</i>	Pointer to the source and destination string for the in-place operation.

len Number of elements in the string.

Description

The functions `ippsUppercase` and `ippsUppercaseLatin` are declared in the `ippch.h` file. These functions convert each alphabetic character of the source string *pSrc* to upper case and stores the result in *pDst*.

The in-place flavors of these functions convert each alphabetic character of the source string *pSrcDst* to upper case and store the result in *pSrcDst*.

The function `ippsUppercase` operates with Unicode characters.

The function `ippsUppercaseLatin` operates with ASCII characters.

[Example 11-5](#) shows how to use the function `ippsUppercaseLatin_8u`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

Lowercase, LowercaseLatin

Converts alphabetic characters of a string to all lowercase symbols.

Syntax

```
ippStatus ippsLowercase_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
ippStatus ippsLowercase_16u_I(Ipp16u* pSrcDst, int len);
ippStatus ippsLowercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
ippStatus ippsLowercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
ippStatus ippsLowercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
ippStatus ippsLowercaseLatin_16u_I(Ipp16u* pSrcDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>pDst</i>	Pointer to the destination string.
<i>pSrcDst</i>	Pointer to the source and destination string for the in-place operation.
<i>len</i>	Number of elements in the string.

Description

The functions `ippsLowercase` and `ippsLowercaseLatin` are declared in the `ippch.h` file. These functions convert each alphabetic character of the source string *pSrc* to lower case and store the result in *pDst*.

The in-place flavors of these functions convert each alphabetic character of the source string *pSrcDst* to lower case and store the result in *pSrcDst*.

The function `ippsLowercase` operates with Unicode characters.

The function `ippsLowercaseLatin` operates with ASCII characters.

[Example 11-5](#) shows how to use the function `ippsLowercaseLatin_8u`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

Example 11-5 Using the functions `ippsUppercaseLatin` and `ippsLowercaseLatin`

```
IppStatus case( void ) {
    IppStatus st;
    char string[] = "These Functions Vary the Case!";
    char copy1[40], copy2[40];
    st = ippsLowercaseLatin_8u((Ipp8u*)string, (Ipp8u*)copy1,
    strlen(string)+1);
    st = ippsUppercaseLatin_8u((Ipp8u*)string, (Ipp8u*)copy2,
    strlen(string)+1);
    printf( "Mixed: %s\n", string );
    printf( "Lower: %s\n", copy1 );
    printf( "Upper: %s\n", copy2 );
    return resOK;
}
```

Example 11-5 Using the functions `ippsUppercaseLatin` and `ippsLowercaseLatin`

```

Output:
Mixed: These Functions Vary the Case!
Lower: these functions vary the case!
Upper: THESE FUNCTIONS VARY THE CASE!

```

Hash

Calculates the hash value for the string.

Syntax

```

IppStatus ippsHash_8u32u(const Ipp8u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHash_16u32u(const Ipp16u* pSrc, int len, Ipp32u* pHashVal);

```

Parameters

<i>pSrc</i>	Pointer to the source string.
<i>len</i>	Number of elements in the string.
<i>pHashVal</i>	Pointer to the result value.

Description

The function `ippsHash` is declared in the `ippch.h` file. This function produces the hash value *pHashVal* for the specified string *pSrc*. The hash value is fairly unique to the given string. If hash values of two strings are different, the strings are different as well. If the hash values are the same, the strings are probably identical. The hash value is calculated in the following manner: initial value is set to 0, then the hash value is calculated successively for each *i*-th string element as $pHashVal[i] = 2 * pHashVal[i-1] \wedge pSrc[i]$, where \wedge denotes a bitwise exclusive OR (XOR) operator. The hash value for the last element is the hash value for the whole string.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

Concat

Concatenates several strings together.

Syntax

```
IppStatus ippsConcat_8u_D2L(const Ipp8u* const pSrc[], const int* pSrcLen[],
    int numSrc, Ipp8u* pDst);

IppStatus ippsConcat_16u_D2L(const Ipp16u* const pSrc[], const int* pSrcLen[],
    int numSrc, Ipp16u* pDst);

IppStatus ippsConcat_8u(const Ipp8u* pSrc1, int len1, const Ipp8u* pSrc2, int
    len2, Ipp8u* pDst);

IppStatus ippsConcat_16u(const Ipp16u* pSrc1, int len1, const Ipp16u* pSrc2,
    int len2, Ipp16u* pDst);
```

Parameters

<i>pSrc1</i>	Pointer to the first source string.
<i>len1</i>	Number of elements in the first string.
<i>pSrc2</i>	Pointer to the second source string.
<i>len2</i>	Number of elements in the second string.
<i>pSrc</i>	Pointer to the array of source strings.
<i>pSrcLen</i>	Pointer to the array of lengths of the source strings.
<i>numSrc</i>	Number of source strings.
<i>pDst</i>	Pointer to the destination string.

Description

The function `ippsConcat` is declared in the `ippch.h` file. This function concatenates several strings together. Functions with `D2L` suffix operate with multiple *numSrc* strings *pSrc[]*, while functions without this suffix operate with two strings *pSrc1* and *pSrc2* only. Resulting string is stored in the *pDst*. Necessary memory blocks should be allocated for the destination string before the function is called. Summary length of the strings to be concatenated can not be greater than `IPP_MAX_32S`.

[Example 11-6](#) shows how to use the function `ippsConcat_8u`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len1</i> or <i>len2</i> is negative, or <i>srcLen[i]</i> is negative for <i>i</i> < <i>numSrc</i> .
<code>ippStsSizeErr</code>	Indicates an error condition if <i>numSrc</i> is equal to or less than 0.

Example 11-6 Using the function `ippsConcat`

```
IppStatus concat( void ) {
    IppStatus st;
    char string[80] = "This is the initial string!";
    char suffix[] = " extra text to add to the string...";
    /* Combine strings with no more than 19 characters of suffix: */
    printf( "Before: %s\n", string );
    st = ippsConcat_8u((Ipp8u*)string, strlen(string), (Ipp8u*)suffix,
strlen(suffix),
    (Ipp8u*)string);
    printf( "After:  %s\n", string );
    return st;
}
```

Output:

Before: This is the initial string!

After: This is the initial string! extra text to add to the string...

ConcatC

Concatenates several strings together and inserts symbol delimiters between them.

Syntax

```
IppStatus ippsConcatC_8u_D2L(const Ipp8u* const pSrc[], const int* pSrcLen[],
    int numSrc, Ipp8u delim, Ipp8u* pDst);
IppStatus ippsConcatC_16u_D2L(const Ipp16u* const pSrc[], const int* pSrcLen[],
    int numSrc, Ipp16u delim, Ipp16u* pDst);
```

Parameters

<i>pSrc</i>	Pointer to the array of source strings.
<i>pSrcLen</i>	Pointer to the array of lengths of the source strings.
<i>numSrc</i>	Number of source strings.
<i>delim</i>	Symbol delimiter.
<i>pDst</i>	Pointer to the destination string.

Description

The function `ippsConcatC` is declared in the `ippch.h` file. This function concatenates *numSrc* strings *pSrc* together and inserts a specified delimiter symbol *delim* between them in the resulting string *pDst*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>srcLen[i]</i> is negative for $i < numSrc$.
<code>ippStsSizeErr</code>	Indicates an error condition if <i>numSrc</i> is equal to or less than 0.

SplitC

Splits source string into separate parts.

Syntax

```
IppStatus ippsSplitC_8u_D2L(const Ipp8u* pSrc, int srcLen, Ipp8u delim,
    Ipp8u* pDst[], int* pDstLen[], int* pNumDst);
IppStatus ippsSplitC_16u_D2L(const Ipp16u* pSrc, int srcLen, Ipp16u delim,
    Ipp16u* pDst[], int* pDstLen[], int* pNumDst);
```

Parameters

<i>pSrc</i>	Pointer to the source strings.
<i>srcLen</i>	Number of elements in the source string.
<i>delim</i>	Symbol delimiter.

<i>pDst</i>	Pointer to the array of the destination strings.
<i>pDstLen</i>	Pointer to array of the destination string lengths.
<i>pNumDst</i>	Number of destination strings.

Description

The function `ippSplitC` is declared in the `ippch.h` file. This function breaks source string *pSrc* into *pNumDst* separate strings *pDst* using a specified symbol *delim* as a delimiter.

If *n* specified delimiters occur in the beginning or in the end of the source string, then *n* empty strings are appended to the array of destination strings.

If *n* specified delimiters occur in a certain position within the source string, then (*n*-1) empty strings will be inserted into the array of destination strings, where *n* is the number of delimiter occurrences in this position.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>srcLen</i> is negative, or <i>dstLen</i> is negative for $i < pNumDst$.
<code>ippStsSizeErr</code>	Indicates an error condition if <i>pNumDst</i> is equal to or less than 0.
<code>ippStsOvermatchStrings</code>	Indicates a warning if number of output strings exceeds the initially specified number <i>pNumDst</i> ; in this case odd strings are discarded.
<code>ippStsOverlongString</code>	Indicates a warning if in some output strings the number of elements exceeds the initially specified value <i>pDstLen</i> ; in this case corresponding strings are truncated to initial lengths.

Regular Expressions

This section describes the Intel IPP functions that perform matching operations with the Perl-compatible regular expression patterns. See <http://search.cpan.org/dist/perl/pod/perlre.pod> for more details about Perl-compatible regular expressions.

The current version of the Intel IPP functions for regular expressions have some limitations, specifically they do not support literal (metacharacters `\l`, `\L`, `\u`, `\U`, `\N{name}`), embedded Perl code (`{code}`), extended regular expression (`??{code}`) and all metacharacters for working with Unicode.

RegExpInitAlloc

Allocates memory and initializes the structure for processing matching operation with regular expressions.

Syntax

```
IppStatus ippsRegExpInitAlloc(const char* pPattern, const char* pOptions,  
    IppRegExpState** ppRegExpState, int* pErrOffset);
```

Parameters

<i>pPattern</i>	Pointer to the pattern of regular expression.
<i>pOptions</i>	Pointer to options for compiling and executing regular expressions (possible values <code>i</code> , <code>s</code> , <code>m</code> , <code>x</code> , <code>g</code>). It must be <code>NULL</code> if no options are required.
<i>ppRegExpState</i>	Double pointer to the structure containing internal form of a regular expression.
<i>pErrOffset</i>	Pointer to the offset in the pattern if compiling is break.

Description

The function `ippsRegExpInitAlloc` is declared in the `ippch.h` file. This function allocates memory and initializes a regular expression state structure *ppRegExpState*. The function compiles the initial pattern of regular expressions *pPattern* in accordance with the compiling options specified by *pOptions*, converts it to the specific internal form, and stores it in the initialized structure. This structure is used by the function [ippsRegExpFind](#) to perform matching operation.

If the compiling is not completed, the function returns the pointer *pErrOffset* pointed to the position in the pattern where the compiling is interrupted.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.
<code>ippStsRegExpOptionsErr</code>	Indicates an error if specified options are incorrect
<code>ippStsRegExpQuantifierErr</code>	Indicates an error if the quantifier is incorrect
<code>ippStsRegExpGroupingErr</code>	Indicates an error if the grouping is incorrect
<code>ippStsRegExpBackRefErr</code>	Indicates an error if the back reference is incorrect
<code>ippStsRegExpChClassErr</code>	Indicates an error if the character class is incorrect
<code>ippStsRegExpMetaChErr</code>	Indicates an error if the metacharacter is incorrect

RegExpFree

Frees the memory allocated for a regular expression state structure.

Syntax

```
IppStatus ippStsRegExpFree(IppsRegExpState* pRegExpState);
```

Parameters

pRegExpState Pointer to the pattern of regular expression.

Description

The function `ippStsRegExpFree` is declared in the `ippch.h` file. This function frees memory allocated by the function [`ippStsRegExpInitAlloc`](#) for the pyramid structure *pRegExpState*.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if <i>pRegExpState</i> pointer is NULL.

RegExpInit

Initializes the structure for processing matching operation with regular expressions.

Syntax

```
IppStatus ippsRegExpInit(const char* pPattern, const char* pOptions,  
    IppRegExpState* pRegExpState, int* pErrOffset)
```

Parameters

<i>pPattern</i>	Pointer to the pattern of regular expression.
<i>pOptions</i>	Pointer to options for compiling and executing regular expressions (possible values i, s, m, x, g). It must be NULL if no options are required.
<i>pRegExpState</i>	Pointer to the structure containing internal form of a regular expression.
<i>pErrOffset</i>	Pointer to the offset in the pattern if compiling is break.

Description

The function `ippsRegExpInit` is declared in the `ippch.h` file. This function initializes a regular expression state structure *pRegExpState* in the external buffer. The size of this buffer should be computed previously by calling the function [ippsRegExpGetSize](#). The function compiles the initial pattern of regular expressions *pPattern* in accordance with the compiling options specified by *pOptions*, converts it to the specific internal form, and stores it in the initialized structure. This structure is used by the function [ippsRegExpFind](#) to perform matching operation.

If the compiling is not completed, the function returns the pointer *pErrOffset* pointed to the position in the pattern where the compiling is interrupted.



CAUTION. The parameter *pPattern* must be the same for both functions `ippsRegExpInit` and `ippsRegExpGetSize`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.

<code>ippStsRegExpOptionsErr</code>	Indicates an error if specified options are incorrect
<code>ippStsRegExpQuantifierErr</code>	Indicates an error if the quantifier is incorrect
<code>ippStsRegExpGroupingErr</code>	Indicates an error if the grouping is incorrect
<code>ippStsRegExpBackRefErr</code>	Indicates an error if the back reference is incorrect
<code>ippStsRegExpChClassErr</code>	Indicates an error if the character class is incorrect
<code>ippStsRegExpMetaChErr</code>	Indicates an error if the metacharacter is incorrect

RegExpGetSize

Computes the size of the regular expression state structure.

Syntax

```
IppStatus ippStsRegExpGetSize(const char* pPattern, int* pRegExpStateSize);
```

Parameters

<code>pPattern</code>	Pointer to the pattern of regular expression.
<code>pRegExpStateSize</code>	Pointer to the computed size of the regular expression structure.

Description

The function `ippStsRegExpGetSize` is declared in the `ippch.h` file. This function computes the size of the memory that is necessary to initialize by the function [ippStsRegExpInit](#) the regular expression state structure containing the pattern `pPattern` in the internal form. The value of the computed size is stored in the `pRegExpStateSize`.



CAUTION. The parameter `pPattern` must be the same for both functions `ippStsRegExpInit` and `ippStsRegExpGetSize`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.
-------------------------------	--

RegExpSetMatchLimit

Sets the value of the `matchLimit` parameter.

Syntax

```
IppStatus ippRegExpSetMatchLimit(int matchLimit, IppRegExpState*  
    pRegExpState);
```

Parameters

<code><i>matchLimit</i></code>	Value of the of the matches kept in stack.
<code><i>pRegExpState</i></code>	Pointer to the structure containing internal form of a regular expression.

Description

The function `ippRegExpSetMatchLimit` is declared in the `ippch.h` file. This function sets the value of the parameter `matchLimit` that specifies how many times the function [ippRegExpFind](#) can be called through the single execution avoiding the possible stack overflow. The default value is set very large, so you should set this parameter to the reasonable value in accordance with your needs.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if the <code><i>pRegExpState</i></code> pointer is NULL.

RegExpFind

Looks for the occurrences of the substrings matching the specified regular expression.

Syntax

```
IppStatus ippsRegExpFind_8u(const Ipp8u* pSrc, int srcLen, IppRegExpState*
    pRegExpState, IppRegExpFind* pFind, int* pNumFind);
```

Parameters

<i>pSrc</i>	Pointer to the source strings.
<i>srcLen</i>	Number of elements in the source string.
<i>pRegExpState</i>	Pointer to the structure containing internal form of regular expression.
<i>pFind</i>	Array of pointers to the matching substrings.
<i>pNumFind</i>	Size of the array <i>pFind</i> on input, number of matching substrings on output.

Description

The function `ippsRegExpFind` is declared in the `ippch.h` file. This function search through the *srcLen* elements of the source string *pSrc* for substrings that match the specified regular expression in accordance with the regular expression pattern that is stored in the structure *pRegExpState*. This structure must be initialized by the functions either [ippsRegExpInit](#) or [ippsRegExpInitAlloc](#) beforehand.

Initially the parameter *pNumFind* specifies the size of array *pFind*, the output parameter *pNumFind* returns the number of the matching substrings.

pFind->pFindOffset specifies the offset of the pointer to the matching substring, and *pFind->lenFind* - number of elements in the matching substring. *pFind[0]* points to the substring that matches the whole regular expression, *pFind[1]* points to the substring that matches the first grouping, *pFind[2]* points to the substring that matches the second grouping, and so on.



CAUTION. It is recommended to set the default value of the parameter *matchLimit* in accordance with real necessity by calling the function [ippsRegExpSetMatchLimit](#) beforehand.

[Example 11-7](#) shows how to use the function `ippRegExpFind_8u`.

Example 11-7 Using of the Function `ippRegExpFind`

```
IppRegExpFind pFind[2];
IppRegExpState *pState;
IppStatus status;
Ipp8u pSrc[] = "Hello World!";
int  srcLen = sizeof(pSrc)/sizeof(pSrc[0]) - 1;
int  num     = sizeof(pFind)/sizeof(pFind[0]);
int  i, err;

// to perform ordinary matching:
if( (status = ippRegExpInitAlloc( "[hH]ello (\\w+)", NULL, &pState, &err )) !=
    ippStsNoErr ) {

// to perform global matching:
if( (status = ippRegExpInitAlloc( "[hH]ello (\\w+)", "g", &pState, &err )) !=
    ippStsNoErr ) {

    printf( "Error: \"%s\" in position %d\\n", ippGetStatusString( status ), err );
    return 1;
}
ippRegExpFind_8u( pSrc, srcLen, pState, pFind, &num );

//Output: ordinary matching:

num = 2;
pFind[0].pFind = "Hello World"; pFind[0].lenFind = 11;
pFind[1].pFind = "World"; pFind[1].lenFind = 5;

//Output: global matching:

num = 1;
pFind[0].pFind = "World"; pFind[0].lenFind = 5;
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if <i>srcLen</i> is negative, or <i>pNumFind</i> is less than or equal to 0.
<code>ippStsRegExpErr</code>	The state structure <i>pRegExpState</i> contains wrong data.
<code>ippStsRegExpMatchLimitErr</code>	The match limit has been exhausted.

Fixed-Accuracy Arithmetic Functions

12

This chapter describes Intel[®] IPP fixed-accuracy transcendental mathematical functions of vector arguments. These functions take an input vector as argument, compute values of the respective elementary function element-wise, and return the results in an output vector.

Function specifications comply with the common API agreement of Intel IPP, but include some new features essential to scientific arithmetic functions. The main feature is a more elaborate specification of accuracy that differs from the common definition in adding several new levels of accuracy, besides original levels introduced by single precision and double precision data formats.

Fixed-accuracy vector functions implementation supports IEEE-754 standard in all flavors, which means that:

- All functions have a precisely determined and guaranteed level of accuracy for all argument values.
- All special value processing and exceptions handling requirements are met, which implies that when accuracy is below the standard level, the function meets IEEE-754 requirements in all other respects.

The choice of accuracy levels should be based on practical experience and identified application demands. Available options are specified in the function name suffix and include A11, A21, or A24 for the single precision, and A50 or A53 for the double precision data format. Flavors A11, A21, and A50 provide approximately 3, 6, and 15 exact decimal digits, respectively. For flavors A24 and A53, the maximum guaranteed error is within 1 ulp and in most cases does not exceed 0.55 ulp.

Fixed-accuracy arithmetic functions subset of Intel IPP has the similar functionality as the respective part of [Intel[®] Math Kernel Library](#) (Intel[®] MKL) .

However, Intel IPP provides lower-level transcendental functions that have separate flavors for each mode of operations and data type and are better suitable for multimedia and signal processing in real time applications.

The full list of these functions is given in [Table 12-1](#).

Table 12-1 Intel IPP Fixed-Accuracy Arithmetic Functions

Function Short Name	Data Types	Description
Power and Root Functions		
Inv	32f, 64f	Computes inverse value of each vector element.
Div	32f, 64f	Divides elements of one vector by corresponding elements of another vector.
Sqrt	32f, 64f	Computes square root of each vector element.
InvSqrt	32f, 64f	Computes inverse square root of each vector element.
Cbrt	32f, 64f	Computes cube root of each vector element.
InvCbrt	32f, 64f	Computes inverse cube root of each vector element.
Pow	32f, 64f	Raises each element of one vector to the power of corresponding element of another vector.
Powx	32f, 64f	Raises each element of a vector to a constant power.
Exponential and Logarithmic Functions		
Exp	32f, 64f	Raises e to the power of each vector element.
Ln	32f, 64f	Computes natural logarithm of each vector element.
Log10	32f, 64f	Computes common logarithm of each vector element.
Trigonometric Functions		
Cos	32f, 64f	Computes cosine of each vector element.
Sin	32f, 64f	Computes sine of each vector element.
SinCos	32f, 64f	Computes sine and cosine of each vector element.
Tan	32f, 64f	Computes tangent of each vector element.
Acos	32f, 64f	Computes inverse cosine of each vector element.
Asin	32f, 64f	Computes inverse sine of each vector element.
Atan	32f, 64f	Computes inverse tangent of each vector element.
Atan2	32f, 64f	Computes four-quadrant inverse tangent of elements of two vectors.
Hyperbolic Functions		
Cosh	32f, 64f	Computes hyperbolic cosine of each vector element.
Sinh	32f, 64f	Computes hyperbolic sine of each vector element.

Table 12-1 Intel IPP Fixed-Accuracy Arithmetic Functions (continued)

Function Short Name	Data Types	Description
Tanh	32f, 64f	Computes hyperbolic tangent of each vector element.
Acosh	32f, 64f	Computes inverse (nonnegative) hyperbolic cosine of each vector element.
Asinh	32f, 64f	Computes inverse hyperbolic sine of each vector element.
Atanh	32f, 64f	Computes inverse hyperbolic tangent of each vector element.
Special Functions		
Erf	32f, 64f	Computes the error function value.
Erfc	32f, 64f	Computes the complementary error function value



NOTE. You should not confuse fixed-accuracy arithmetic functions described in this chapter with functions in chapter 5 that have similar functionality but follow different accuracy specifications.

Intel IPP fixed-accuracy arithmetic functions may return status codes of normal execution (IppStsNoErr), error conditions IppStsSizeErr, IppStsNullPtrErr (see [Table 2-3](#) in Chapter 2), and the following specific warnings: IppStsDomain, IppStsSingularity, IppStsOverflow, IppStsUnderflow. In case of warnings, the value returned is positive and the computation is continued.

[Table 12-2](#) lists status codes and the corresponding messages for these warnings.

Table 12-2 Warning Status Codes for Fixed-Accuracy Arithmetic Functions

Status	Value	Message
IppStsOverflow	12	Overflow occurred in the operation.
IppStsUnderflow	17	Underflow occurred in the operation.
IppStsSingularity	18	Singularity occurred in the operation.
IppStsDomain	19	Argument is out of the function domain.

See [“Handling of Special Cases”](#) for more information on function operation in cases when their arguments take on specific values that are outside the range of function definition.

Power and Root Functions

Inv

*Computes inverse value
of each vector element.*

Syntax

```
IppStatus ippInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);  
IppStatus ippInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);  
IppStatus ippInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);  
IppStatus ippInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);  
IppStatus ippInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippInv` is declared in the `ippvm.h` file. This function computes the inverse value of each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

- function flavor `ippInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

- function flavor `ippInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

- function flavor `ippInv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{pSrc[n]}, 0 \leq n < len.$$

[Example 12-1](#) shows how to use the function `ippsInv`.

Example 12-1 Using `ippsInv` Function

```
IppStatus ippsInv_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-9.975, 1.272, -6.134, 6.175};
    Ipp32f      y[4];

    IppStatus st = ippsInv_32f_A21( x, y, 4 );

    printf(" ippsInv_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsInv_32f_A21:
x = -9.975 1.272 -6.134 6.175
y = -0.100 0.786 -0.163 0.162
```

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is null.
<code>ippsStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of <code>pSrc</code> is equal to 0.

Div

*Divides each element of the first vector
by corresponding element of the second vector.*

Syntax

```
IppStatus ippsDiv_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,  
    Ipp32f* pDst, int len);  
IppStatus ippsDiv_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,  
    Ipp32f* pDst, int len);  
IppStatus ippsDiv_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,  
    Ipp32f* pDst, int len);  
IppStatus ippsDiv_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,  
    Ipp64f* pDst, int len);  
IppStatus ippsDiv_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,  
    Ipp64f* pDst, int len);
```

Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsDiv` is declared in the `ippvm.h` file. This function divides each element of the vector *pSrc1* by the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsDiv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsDiv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsDiv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsDiv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsDiv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as $pDst[n] = \frac{pSrc1[n]}{pSrc2[n]}, 0 \leq n < len$.

[Example 12-2](#) shows how to use the function `ippsDiv`.

Example 12-2 Using `ippsDiv` Function

```
IppStatus ippsDiv_32f_A21_sample(void)
{
    const Ipp32f x1[4] = {599.088, 735.034, 572.448, 151.640};
    const Ipp32f x2[4] = {385.297, 609.005, 361.403, 225.182};
    Ipp32f      y[4];

    IppStatus st = ippsDiv_32f_A21( x1, x2, y, 4 );

    printf(" ippsDiv_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2],
x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2],
x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],
y[3]);
    return st;
}
```

Output results:

```
ippsDiv_32f_A21:
x1 = 599.088 735.034 572.448 151.640
x2 = 385.297 609.005 361.403 225.182
y  = 1.555 1.207 1.584 0.673
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc1</i> or <i>pSrc2</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of <i>pSrc2</i> is equal to 0.

Sqrt

Computes square root of each vector element.

Syntax

```
IppStatus ippsSqrt_32f_A11 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_32f_A21 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_32f_A24 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_64f_A50 ( const Ipp64f*  pSrc, Ipp64f*  pDst, int len );
IppStatus ippsSqrt_64f_A53 ( const Ipp64f*  pSrc, Ipp64f*  pDst, int len );
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsSqrt` is declared in the `ippvm.h` file. This function computes square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsSqrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsSqrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsSqrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsSqrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsSqrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \sqrt{pSrc[n]}, 0 \leq n < len.$$

[Example 12-3](#) shows how to use the function `ippsSqrt`.

Example 12-3 Using `ippsSqrt` Function

```

IppStatus ippsSqrt_32f_A21_sample(void)
{
    const Ipp32f x[4] = {5850.093, 4798.730, 3502.915, 8959.624};
    Ipp32f        y[4];

    IppStatus st = ippsSqrt_32f_A21( x, y, 4 );

    printf(" ippsSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsSqrt_32f_A21:
x = 5850.093 4798.730 3502.915 8959.624
y = 76.486 69.273 59.185 94.655

```

Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> is less than 0.

InvSqrt

Computes inverse square root of each vector element.

Syntax

```
IppStatus ippsInvSqrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInvSqrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsInvSqrt` is declared in the `ippvm.h` file. This function computes inverse square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

- function flavor `ippsInvSqrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;
- function flavor `ippsInvSqrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;
- function flavor `ippsInvSqrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInvSqrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInvSqrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{\sqrt{pSrc[n]}}, 0 \leq n < len.$$

[Example 12-4](#) shows how to use the function `ippsInvSqrt`.

Example 12-4 Using `ippsInvSqrt` Function

```

IppStatus ippsInvSqrt_32f_A21_sample(void)
{
    const Ipp32f x[4] = {7105.043, 5135.398, 3040.018, 149.944};
    Ipp32f      y[4];

    IppStatus st = ippsInvSqrt_32f_A21( x, y, 4 );

    printf(" ippsInvSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsInvSqrt_32f_A21:
x = 7105.043 5135.398 3040.018 149.944
y = 0.012 0.014 0.018 0.082

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is less than 0.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of <code>pSrc</code> is equal to 0.

Cbrt

Computes cube root of each vector element.

Syntax

```
IppStatus ippsCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsCbrt` is declared in the `ippvm.h` file. This function computes cube root of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsCbrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCbrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCbrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \sqrt[3]{pSrc[n]}, 0 \leq n < len.$$

[Example 12-5](#) shows how to use the function `ippsCbrt`.

Example 12-5 Using `ippsCbrt` Function

```
IppStatus ippsCbrt_32f_A21_sample(void)
{
    const Ipp32f x[4] = {6456.801, 4932.096, -6517.838, 7178.869};
    Ipp32f        y[4];

    IppStatus st = ippsCbrt_32f_A21( x, y, 4 );

    printf(" ippsCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsCbrt_32f_A21:
x = 6456.801 4932.096 -6517.838 7178.869
y = 18.621 17.022 -18.680 19.291
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

InvCbrt

Computes inverse cube root of each vector element.

Syntax

```
IppStatus ippsInvCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

```
IppStatus ippsInvCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsInvCbrt` is declared in the `ippvm.h` file. This function computes inverse cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

- function flavor `ippsInvCbrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;
- function flavor `ippsInvCbrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;
- function flavor `ippsInvCbrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

- function flavor `ippsInvCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;
- function flavor `ippsInvCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{\sqrt[3]{pSrc[n]}}, 0 \leq n < len.$$

[Example 12-6](#) shows how to use the function `ippsInvCbrt`.

Example 12-6 Using ippsInvCbrt Function

```
IppStatus ippsInvCbrt_32f_A21_sample(void)
{
    const Ipp32f x[4] = {914.120, 3644.584, 1473.214, 1659.070};
    Ipp32f      y[4];

    IppStatus st = ippsInvCbrt_32f_A21( x, y, 4 );
}
```

Example 12-6 Using `ippsInvCbrt` Function (continued)

```

    printf(" ippsInvCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsInvCbrt_32f_A21:
x = 914.120 3644.584 1473.214 1659.070
y = 0.103 0.065 0.088 0.084

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of <i>pSrc</i> is equal to 0.

Pow

Raises each element of the first vector to the power of corresponding element of the second vector.

Syntax

```

IppStatus ippsPow_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);

```

```
IppStatus ippsPow_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
```

Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsPow` is declared in the `ippvm.h` file. This function raises each element of vector *pSrc1* to the power of the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsPow_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsPow_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsPow_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsPow_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsPow_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc1[n])^{pSrc2[n]}, 0 \leq n < len.$$

[Example 12-7](#) shows how to use the function `ippsPow`.

Example 12-7 Using `ippsPow` Function

```
IppStatus ippsPow_32f_A21_sample(void)
{
```

Example 12-7 Using `ippsPow` Function (continued)

```

const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32f x2[4] = {0.823, 0.991, 0.411, 0.692};
Ipp32f      y[4];

    IppStatus st = ippsPow_32f_A21( x1, x2, y, 4 );

    printf(" ippsPow_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2],
x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2],
x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],
y[3]);
    return st;
}

```

Output results:

```

ippsPow_32f_A21:
x1 = 0.483 0.565 0.776 0.252
x2 = 0.823 0.991 0.411 0.692
y  = 0.549 0.568 0.901 0.386

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc1</i> or <i>pSrc2</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one pair of the source elements meets the following condition: element of <i>pSrc1</i> is finite, less than 0, and element of <i>pSrc2</i> is finite, non-integer.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of <i>pSrc1</i> is equal to 0, and element of <i>pSrc2</i> is integer and less than 0.

Powx

*Raises each element of a vector
to the constant power.*

Syntax

```
IppStatus ippsPowx_32f_A11 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
    Ipp32f* pDst, int len);
IppStatus ippsPowx_32f_A21 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
    Ipp32f* pDst, int len);
IppStatus ippsPowx_32f_A24 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
    Ipp32f* pDst, int len);
IppStatus ippsPowx_64f_A50 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
    Ipp64f* pDst, int len);
IppStatus ippsPowx_64f_A53 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
    Ipp64f* pDst, int len);
```

Parameters

<i>pSrc1</i>	Pointer to the source vector.
<i>ConstValue</i>	Constant value.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsPowx` is declared in the `ippvm.h` file. This function raises each element of the vector *pSrc1* to the constant power *ConstValue* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsPowx_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsPowx_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsPowx_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsPowx_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsPowx_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc1[n])^{ConstValue}, 0 \leq n < len.$$

[Example 12-8](#) shows how to use the function `ippsPowx`.

Example 12-8 Using `ippsPowx` Function

```
IppStatus ippsPowx_32f_A21_sample(void)
{
    const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
    const Ipp32f x2 = 0.823;
    Ipp32f      y[4];

    IppStatus st = ippsPowx_32f_A21( x1, x2, y, 4 );

    printf(" ippsPowx_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2],
x1[3]);
    printf(" x2 = %.3f \n", x2);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2],
y[3]);
    return st;
}
```

Output results:

```
ippsPowx_32f_A21:
x1 = 0.483 0.565 0.776 0.252
x2 = 0.823
y  = 0.549 0.568 0.901 0.386
```

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when `pSrc1` or `pDst` pointer is null.

<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one pair of the elements meets the following condition: element of <i>pSrc1</i> is finite, less than 0, and <i>ConstValue</i> is finite, non-integer.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of <i>pSrc1</i> is equal to 0, and <i>ConstValue</i> is integer and less than 0.

Exponential and Logarithmic Functions

Exp

Raises e to the power of each vector element.

Syntax

```
IppStatus ippsExp_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsExp` is declared in the `ippvm.h` file. This function raises e to the power of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsExp_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsExp_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsExp_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsExp_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippExp_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = e^{pSrc[n]}, 0 \leq n < len.$$

[Example 12-9](#) shows how to use the function `ippExp`.

Example 12-9 Using `ippExp` Function

```

IppStatus ippExp_32f_A21_sample(void)
{
    const Ipp32f x[4] = {4.885, -0.543, -3.809, -4.953};
    Ipp32f      y[4];

    IppStatus st = ippExp_32f_A21( x, y, 4 );

    printf(" ippExp_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippExp_32f_A21:
x = 4.885 -0.543 -3.809 -4.953
y = 132.324 0.581 0.022 0.007

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsOverflow</code>	Indicates a warning that the function overflows, that is, at least one of elements of <i>pSrc</i> is greater than $\text{Ln}(\text{FPMAX})$, where FPMAX is the maximum representable floating-point number.
<code>IppStsUnderflow</code>	Indicates a warning that the function underflows, that is, at least one of elements of <i>pSrc</i> is less than $\text{Ln}(\text{FPMIN})$, where FPMIN is the minimum positive floating-point value.

Ln

*Computes natural logarithm
of each vector element.*

Syntax

```
IppStatus ippsLn_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsLn` is declared in the `ippvm.h` file. This function computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsLn_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsLn_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsLn_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsLn_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsLn_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \log_e(pSrc[n]), 0 \leq n < len.$$

[Example 12-10](#) shows how to use the function `ippsLn`.

Example 12-10 Using `ippsLn` Function

```
IppStatus ippsLn_32f_A21_sample(void)
{
    const Ipp32f x[4] = {0.188, 3.841, 5.363, 5.755};
    Ipp32f      y[4];

    IppStatus st = ippsLn_32f_A21( x, y, 4 );

    printf(" ippsLn_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsLn_32f_A21:
x = 0.188 3.841 5.363 5.755
y = -1.670 1.346 1.680 1.750
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is less than 0.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of <code>pSrc</code> is equal to 0.

Log10

*Computes common logarithm
of each vector element.*

Syntax

```
IppStatus ippsLog10_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLog10_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsLog10` is declared in the `ippvm.h` file. This function computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsLog10_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsLog10_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsLog10_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsLog10_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsLog10_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \log_{10}(pSrc[n]), 0 \leq n < len.$$

[Example 12-11](#) shows how to use the function `ippsLog10`.

Example 12-11 Using `ippsLog10` Function

```

IppStatus ippsLog10_32f_A21_sample(void)
{
    const Ipp32f x[4] = {6.057, 6.111, 1.746, 6.664};
    Ipp32f      y[4];

    IppStatus st = ippsLog10_32f_A21( x, y, 4 );

    printf(" ippsLog10_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsLog10_32f_A21:
x = 6.057 6.111 1.746 6.664
y = 0.782 0.786 0.242 0.824

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> is less than 0.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of <i>pSrc</i> is equal to 0.

Trigonometric Functions

Cos

Computes cosine of each vector element.

Syntax

```
IppStatus ippsCos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsCos` is declared in the `ippvm.h` file. This function computes a cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

- function flavor `ippsCos_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

- function flavor `ippsCos_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

- function flavor `ippsCos_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCos_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCos_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \cos(pSrc[n]), 0 \leq n < len.$$

[Example 12-12](#) shows how to use the function `ippsCos`.

Example 12-12 Using `ippsCos` Function

```

IppStatus ippsCos_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-984.222, -2957.549, -8859.218, 2153.691};
    Ipp32f      y[4];

    IppStatus st = ippsCos_32f_A21( x, y, 4 );

    printf(" ippsCos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsCos_32f_A21:
x = -984.222 -2957.549 -8859.218 2153.691
y = -0.619 -0.258 0.997 0.129

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is equal to $\pm \text{INF}$.

Sin

Computes sine of each vector element.

Syntax

```
IppStatus ippsSin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsSin` is declared in the `ippvm.h` file. This function computes a sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsSin_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsSin_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsSin_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsSin_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsSin_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \sin(pSrc[n]), 0 \leq n < len.$$

[Example 12-13](#) shows how to use the function `ippSin`.

Example 12-13 Using `ippSin` Function

```
IppStatus ippSin_32f_A21_sample(void)
{
    const Ipp32f x[4] = {5666.372, 6052.125, 397.656, -3960.997};
    Ipp32f      y[4];

    IppStatus st = ippSin_32f_A21( x, y, 4 );

    printf(" ippSin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippSin_32f_A21:
x = 5666.372 6052.125 397.656 -3960.997
y = -0.873 0.988 0.970 -0.524
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is equal to $\pm \text{INF}$.

SinCos

Computes sine and cosine of each vector element.

Syntax

```
IppStatus ippSinCos_32f_A11 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
```

```

IppStatus ippsSinCos_32f_A21 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_32f_A24 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A50 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A53 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);

```

Parameters

<i>pSrc</i>	Pointer to the first source vector.
<i>pDst1</i>	Pointer to the destination vector for sine values.
<i>pDst2</i>	Pointer to the destination vector for cosine values.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsSinCos` is declared in the `ippvm.h` file. This function computes sine of each element of *pSrc* and stores the result in the corresponding element of *pDst1*; computes cosine of each element of *pSrc* and stores the result in the corresponding element of *pDst2*.

For single precision data:

function flavor `ippsSinCos_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsSinCos_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsSinCos_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsSinCos_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsSinCos_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst1[n] = \sin(pSrc[n]), pDst2[n] = \cos(pSrc[n]), 0 \leq n < len.$$

[Example 12-14](#) shows how to use the function `ippsSinCos`.

Example 12-14 Using `ippsSinCos` Function

```
IppStatus ippsSinCos_32f_A21_sample(void)
{
    const Ipp32f x[4] = {3857.845, -3939.024, -1468.856, -8592.486};
    Ipp32f      y1[4];
    Ipp32f      y2[4];

    IppStatus st = ippsSinCos_32f_A21( x, y1, y2, 4 );

    printf(" ippsSinCos_32f_A21:\n");
    printf(" x  = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2],
x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2],
y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2],
y2[3]);
    return st;
}
```

Output results:

```
ippsSinCos_32f_A21:
x  = 3857.845 -3939.024 -1468.856 -8592.486
y1 = -0.031 0.508 0.987 0.228
y2 = 1.000 0.861 0.161 -0.974
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDst1</i> or <i>pDst2</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the <i>pSrc</i> elements is equal to $\pm \text{INF}$.

Tan

Computes tangent of each vector element.

Syntax

```
IppStatus ippsTan_32f_All (const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```

IppStatus ippTan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippTan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippTan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippTan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippTan` is declared in the `ippvm.h` file. This function computes the tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

- function flavor `ippTan_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;
- function flavor `ippTan_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;
- function flavor `ippTan_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

- function flavor `ippTan_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;
- function flavor `ippTan_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows: $pDst[n] = \tan(pSrc[n]), 0 \leq n < len$.

[Example 12-15](#) shows how to use the function `ippTan`.

Example 12-15 Using `ippTan` Function

```

IppStatus ippTan_32f_A21_sample(void)
{
    const Ipp32f x[4] = {7519.456, 4533.524, 9118.015, 8514.359};
    Ipp32f      y[4];

```

Example 12-15 Using `ippsTan` Function (continued)

```

IppStatus st = ippsTan_32f_A21( x, y, 4 );

printf(" ippsTan_32f_A21:\n");
printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
return st;
}

```

Output results:

```

ippsTan_32f_A21:
x = 7519.456 4533.524 9118.015 8514.359
y = -18.656 0.209 2.028 0.750

```

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> is equal to $\pm \text{INF}$.

Acos

Computes inverse cosine of each vector element.

Syntax

```

IppStatus ippsAcos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.

len Number of elements in the vectors.

Description

The function `ippsAcos` is declared in the `ippvm.h` file. This function computes the inverse cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsAcos_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAcos_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsAcos_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAcos_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAcos_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \text{acos}(pSrc[n]), 0 \leq n < len.$$

[Example 12-16](#) shows how to use the function `ippsAcos`.

Example 12-16 Using `ippsAcos` Function

```
IppStatus ippsAcos_32f_A21_sample(void)
{
    const Ipp32f x[4] = {0.079, -0.715, -0.076, -0.529};
    Ipp32f      y[4];

    IppStatus st = ippsAcos_32f_A21( x, y, 4 );

    printf(" ippsAcos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

ippsAcos_32f_A21:

Example 12-16 Using `ippsAcos` Function (continued)

```
x = 0.079 -0.715 -0.076 -0.529
y = 1.492 2.368 1.647 2.129
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> has an absolute value greater than 1.

Asin

Computes inverse sine of each vector element.

Syntax

```
IppStatus ippsAsin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsAsin` is declared in the `ippvm.h` file. This function computes the inverse sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsAsin_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAsin_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsAsin_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAsin_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAsin_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows: $pDst[n] = \text{asin}(pSrc[n])$, $0 \leq n < len$.

[Example 12-17](#) shows how to use the function `ippsAsin`.

Example 12-17 Using `ippsAsin` Function

```

IppStatus ippsAsin_32f_A21_sample(void)
{
    const Ipp32f x[4] = {0.724, -0.581, 0.559, 0.687};
    Ipp32f      y[4];

    IppStatus st = ippsAsin_32f_A21( x, y, 4 );

    printf(" ippsAsin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsAsin_32f_A21:
x = 0.724 -0.581 0.559 0.687
y = 0.810 -0.620 0.594 0.758

```

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when `pSrc` or `pDst` pointer is null.

<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> has an absolute value greater than 1.

Atan

Computes inverse tangent of each vector element.

Syntax

```
IppStatus ippAtan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippAtan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippAtan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippAtan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippAtan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>len</code>	Number of elements in the vectors.

Description

The function `ippAtan` is declared in the `ippvm.h` file. This function computes the inverse tangent of each element of `pSrc`, and stores the result in the corresponding element of `pDst`.

For single precision data:

function flavor `ippAtan_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippAtan_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippAtan_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAtan_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAtan_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \text{atan}(pSrc[n]), 0 \leq n < len.$$

[Example 12-18](#) shows how to use the function `ippsAtan`.

Example 12-18 Using `ippsAtan` Function

```

IppStatus ippsAtan_32f_A21_sample(void)
{
    const Ipp32f x[4] = {0.994, 0.999, 0.223, -0.215};
    Ipp32f      y[4];

    IppStatus st = ippsAtan_32f_A21( x, y, 4 );

    printf(" ippsAtan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsAtan_32f_A21:
x = 0.994 0.999 0.223 -0.215
y = 0.782 0.785 0.219 -0.212

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

```
IppStatus ippsAtan2_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
IppStatus ippsAtan2_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
```

Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsAtan2` is declared in the `ippvm.h` file. This function computes the angle between the X axis and the line from the origin to the point (X,Y), for each element of *pSrc1* as a Y (the ordinate) and corresponding element of *pSrc2* as an X (the abscissa), and stores the result in the corresponding element of *pDst*. The result angle varies from $-\pi$ to $+\pi$.

For single precision data:

function flavor `ippsAtan2_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAtan2_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsAtan2_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAtan2_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAtan2_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \arctan2(pSrc1[n], pSrc2[n]) \quad , 0 \leq n < len.$$

[Example 12-19](#) shows how to use the function `ippsAtan2`.

Example 12-19 Using `ippsAtan2` Function

```

IppStatus ippsAtan2_32f_A21_sample(void)
{
    const Ipp32f x1[4] = {1.492, 1.700, 1.147, 1.142};
    const Ipp32f x2[4] = {1.064, 1.505, 1.950, 1.905};
    Ipp32f      y[4];

    IppStatus st = ippsAtan2_32f_A21( x1, x2, y, 4 );

    printf(" ippsAtan2_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2],
x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2],
x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2],
y[3]);
    return st;
}

```

Output results:

```

ippsAtan2_32f_A21:
x1 = 1.492 1.700 1.147 1.142
x2 = 1.064 1.505 1.950 1.905
y  = 0.951 0.846 0.532 0.540

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc1</i> or <i>pSrc2</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Hyperbolic Functions

Cosh

*Computes hyperbolic cosine
of each vector element.*

Syntax

```
IppStatus ippsCosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsCosh` is declared in the `ippvm.h` file. This function computes the hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

- function flavor `ippsCosh_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

- function flavor `ippsCosh_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

- function flavor `ippsCosh_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCosh_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCosh_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \cosh(pSrc[n]), 0 \leq n < len.$$

[Example 12-20](#) shows how to use the function `ippsCosh`.

Example 12-20 Using `ippsCosh` Function

```
IppStatus ippsCosh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-4.676, -4.054, 6.803, -9.525};
    Ipp32f      y[4];

    IppStatus st = ippsCosh_32f_A21( x, y, 4 );

    printf(" ippsCosh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsCosh_32f_A21:
x = -4.676 -4.054 6.803 -9.525
y = 53.661 28.833 450.219 6849.870
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsOverflow</code>	Indicates a warning that the function overflows, that is, at least one of elements of <i>pSrc</i> has the absolute value greater than $\text{Ln}(\text{FPMAX}) + \text{Ln}(2)$, where FPMAX is the maximum representable floating-point number.

Sinh

Computes hyperbolic sine of each vector element.

Syntax

```
IppStatus ippsSinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsSinh` is declared in the `ippvm.h` file. This function computes the hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsSinh_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsSinh_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsSinh_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsSinh_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsSinh_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \sinh(pSrc[n]), 0 \leq n < len.$$

[Example 12-21](#) shows how to use the function `ippsSinh`.

Example 12-21 Using `ippsSinh` Function

```

IppStatus ippsSinh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-2.483, -8.148, 3.544, -8.876};
    Ipp32f      y[4];

    IppStatus st = ippsSinh_32f_A21( x, y, 4 );

    printf(" ippsSinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsSinh_32f_A21:
x = -2.483 -8.148 3.544 -8.876
y = -5.945 -1727.412 17.290 -3577.970

```

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppsStsOverflow</code>	Indicates a warning that the function overflows, that is, at least one of elements of <i>pSrc</i> has the absolute value greater than $\text{Ln}(\text{FPMAX}) + \text{Ln}(2)$, where <i>FPMAX</i> is the maximum representable floating-point number.

Tanh

*Computes hyperbolic tangent
of each vector element.*

Syntax

```
IppStatus ippsTanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsTanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsTanh` is declared in the `ippvm.h` file. This function computes the hyperbolic tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsTanh_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsTanh_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsTanh_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsTanh_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsTanh_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[x] = \tanh(pSrc[n]), 0 \leq n < len.$$

[Example 12-22](#) shows how to use the function `ippsTanh`.

Example 12-22 Using `ippsTanh` Function

```
IppStatus ippsTanh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
    Ipp32f      y[4];

    IppStatus st = ippsTanh_32f_A21( x, y, 4 );

    printf(" ippsTanh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsTanh_32f_A21:
x = -0.982 0.838 -0.448 -0.454
y = -0.754 0.685 -0.420 -0.425
```

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is null.
<code>ippsStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

Acosh

Computes inverse (nonnegative) hyperbolic cosine of each vector element.

Syntax

```
IppStatus ippsAcosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```

IppStatus ippsAcosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsAcosh` is declared in the `ippvm.h` file. This function computes the inverse (nonnegative) hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsAcosh_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAcosh_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsAcosh_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAcosh_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAcosh_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[x] = \operatorname{acosh}(pSrc[n]), 0 \leq n < len.$$

[Example 12-23](#) shows how to use the function `ippsAcosh`.

Example 12-23 Using `ippsAcosh` Function

```

IppStatus ippsAcosh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {588.321, 691.492, 837.773, 726.767};

```

Example 12-23 Using `ippsAcosh` Function (continued)

```
Ipp32f      y[4];

IppStatus st = ippsAcosh_32f_A21( x, y, 4 );

printf(" ippsAcosh_32f_A21:\n");
printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
return st;
}
```

Output results:

```
ippsAcosh_32f_A21:
x = 588.321 691.492 837.773 726.767
y = 7.070 7.232 7.424 7.282
```

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> is less than 1.

Asinh

Computes inverse hyperbolic sine of each vector element.

Syntax

```
IppStatus ippsAsinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsAsinh` is declared in the `ippvm.h` file. This function computes the inverse hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsAsinh_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAsinh_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsAsinh_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAsinh_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAsinh_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \sinh(pSrc[n]), 0 \leq n < len.$$

[Example 12-24](#) shows how to use the function `ippsAsinh`.

Example 12-24 Using `ippsAsinh` Function

```

IppStatus ippsAsinh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-30.122, -589.282, 487.472, -63.082};
    Ipp32f      y[4];

    IppStatus st = ippsAsinh_32f_A21( x, y, 4 );

    printf(" ippsAsinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
}

```


Example 12-24 Using `ippsAsinh` Function (continued)

```
        printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
        return st;
    }
```

Output results:

```
ippsAsinh_32f_A21:
x = -30.122 -589.282 487.472 -63.082
y = -4.099 -7.072 6.882 -4.838
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Atanh

Computes inverse hyperbolic tangent of each vector element.

Syntax

```
IppStatus ippsAtanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAtanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsAtanh` is declared in the `ippvm.h` file. This function computes the inverse hyperbolic tangent of each element of `pSrc`, and stores the result in the corresponding element of `pDst`.

For single precision data:

function flavor `ippsAtanh_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAtanh_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsAtanh_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAtanh_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAtanh_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \operatorname{atanh}(pSrc[n]), 0 \leq n < len.$$

[Example 12-25](#) shows how to use the function `ippsAtanh`.

Example 12-25 Using `ippsAtanh` Function

```
IppStatus ippsAtanh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-0.076, 0.808, 0.440, -0.705};
    Ipp32f      y[4];

    IppStatus st = ippsAtanh_32f_A21( x, y, 4 );

    printf(" ippsAtanh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsAtanh_32f_A21:
x = -0.076 0.808 0.440 -0.705
```

Example 12-25 Using `ippAtanh` Function (continued)

```
y = -0.076 1.123 0.472 -0.877
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> has absolute value greater than 1.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of <i>pSrc</i> has absolute value equal to 1.

Special Functions

Erf

Computes the error function value.

Syntax

```
IppStatus ippErf_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippErf_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippErf_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippErf_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippErf_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsErf` is declared in the `ippvm.h` file. This function computes the error function value for each element of `pSrc` and stores the result in the corresponding element of `pDst`.

For single precision data:

function flavor `ippsErf_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErf_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErf_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErf_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErf_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \text{erf}(pSrc[n]), 0 \leq n < len, \text{ where } \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

[Example 12-26](#) shows how to use the function `ippsErf`.

Example 12-26 Using `ippsErf` Function

```
IppStatus ippsErf_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
    Ipp32f      y[4];

    IppStatus st = ippsErf_32f_A21( x, y, 4 );

    printf(" ippsErf_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

Example 12-26 Using `ippsErf` Function (continued)

```
ippsErf_32f_A21:
x = -0.982 0.838 -0.448 -0.454
y = -0.754 0.685 -0.420 -0.425
```

Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

Erfc

Computes the complementary error function value.

Syntax

```
IppStatus ippsErfc_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsErfc_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsErfc` is declared in the `ippvm.h` file. This function computes the complementary error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErfc_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErfc_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErfc_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErfc_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErfc_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \operatorname{erfc}(pSrc[n]), 0 \leq n < len, \text{ where } \operatorname{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

[Example 12-27](#) shows how to use the function `ippsErfc`.

Example 12-27 Using `ippsErfc` Function

```

IppStatus ippsErfc_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
    Ipp32f      y[4];

    IppStatus st = ippsErfc_32f_A21( x, y, 4 );

    printf(" ippsErfc_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsErfc_32f_A21:
x = -0.982 0.838 -0.448 -0.454
y = -0.754 0.685 -0.420 -0.425

```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is null.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsUnderflow</code>	Indicates a warning that the function underflows, that is, at least one of <i>pSrc</i> elements is less than some threshold value, where the function result is less than the minimum positive floating-point value in target precision.

Data Compression Functions

13

This chapter describes Intel[®] IPP functions for data compression that support a number of different compression methods: Huffman and variable-length coding, dictionary-based coding methods (including support of ZLIB compression), and methods based on Burrows-Wheeler Transform.

These functions are grouped into the following sections:

[VLC and Huffman Coding Functions](#)

[Dictionary-Based Compression Functions](#)

[BWT-Based Compression Functions](#)

Each section starts with a table that lists functions described in more detail later in this section, together with the brief description of operations that these functions perform.

Application Notes

- The functions in this domain can be divided into two types: the functions that actually compress data, and transformation functions. The latter do not compress data but only modify them and prepare for further compression. The examples of such transformation are the Burrows-Wheeler Transform, or MoveToFront algorithm. To do data compression efficiently, you should develop the proper consequence of functions of different type that will transform data and then compress them.
- Compression ratio depends on the statistics of input data. For some types of input data no compression could be achieved at all.
- The size of memory required for the output of data compression functions typically is not obvious. As a rule encoding functions use less memory for output than the size of the input buffer, on the contrary decoding functions use more memory for output than the size of the input buffer. You should account these issues and allocate the proper quantity of output

memory using the techniques provided by functions in this domain. For example, you can use a double-pointer technique for automatic shifting the user submitted pointer. For some other functions it is possible to compute the upper limit of the size of the required output buffer.

VLC and Huffman Coding Functions

This section describes the Intel IPP functions for Huffman and variable-length coding.

The full list of these functions is given in [Table 13-1](#).

Table 13-1 Intel IPP VLC and Huffman Coding Functions

Function Short Name	Description
VLC Functions	
<u>VLCEncodeInitAlloc</u>	Allocates memory and initializes the VLC encoder structure.
<u>VLCEncodeFree</u>	Frees memory allocated for the VLC encoder structure.
<u>VLCEncodeInit</u>	Initializes the VLC encoder structure.
<u>VLCEncodeGetSize</u>	Computes the size of the VLC encode structure.
<u>VLCEncodeBlock</u>	Performs VLC encoding of a block of data.
<u>VLCEncodeOne</u>	Performs VLC encoding of a single element.
<u>VLCCountBits</u>	Computes a number of bits required to encode the source block.
<u>VLCDecodeInitAlloc</u>	Allocates memory and initializes the VLC decoder structure.
<u>VLCDecodeFree</u>	Frees memory allocated for the VLC decoder structure.
<u>VLCDecodeInit</u>	Initializes the VLC decoder structure.
<u>VLCDecodeGetSize</u>	Computes the size of the VLC decoder structure.
<u>VLCDecodeBlock</u>	Decodes a block of VLC encoded data.
<u>VLCDecodeOne</u>	Decodes a single VLC encoded element.
Huffman Encoding and Decoding Functions	
<u>EncodeHuffInitAlloc</u>	Allocates memory and initializes the Huffman encoding structure.
<u>HuffFree</u>	Frees memory allocated for the Huffman coding state structures.
<u>EncodeHuffInit</u>	Initializes the Huffman encoding structure.
<u>HuffGetSize</u>	Computes size of the external buffer for the Huffman encoding structure.
<u>EncodeHuffOne</u>	Performs Huffman encoding of a single element.
<u>EncodeHuff</u>	Performs Huffman encoding.
<u>EncodeHuffFinal</u>	Performs Huffman encoding of the remainder.
<u>HuffGetLenCodeTable</u>	Returns the table containing lengths of Huffman codes.
<u>DecodeHuffInitAlloc</u>	Allocates memory and initializes the Huffman decoding structure.
<u>DecodeHuffInit</u>	Initializes the Huffman decoding structure.
<u>DecodeHuffOne</u>	Performs Huffman decoding of a single code word.

Table 13-1 Intel IPP VLC and Huffman Coding Functions (continued)

Function Short Name	Description
DecodeHuff	Performs Huffman decoding.
HuffGetDstBuffSize	Computes the size of a destination buffer for Huffman encoding and decoding by blocks
HuffLenCodeTablePack	Packs the table containing lengths of Huffman codes.
HuffLenCodeTableUnpack	Unpacks the table containing lengths of Huffman codes.

Data Compression VLC functions

Variable Length Coding (VLC) is a data compression method that uses statistical modeling to define which values occur more frequently than others to build tables for subsequent encoding and decoding operations. Data in the bitstream is encoded with VLC table so that the shortest codes correspond to the most frequent values and the longer codes correspond to the less frequent values Intel IPP functions use the structure `IppsVLCTable` for storing the VLC table. The [Table 13-2](#) lists the fields of this structure.

Table 13-2 IppsVLCTable Structure

```
struct VLCTable_32s
{
    Ipp32s value,           The current code.
    Ipp32s code,           The actual bit code for the value.
    Ipp32s length;        The bit length of the value.
}

typedef struct VLCTable_32s IppsVLCTable_32s
```

Fill in the table with codes and code lengths for values, specifying them in the corresponding fields of the `IppsVLCTable` structure. The value can be both positive and negative. The code denotes the length least significant bits in binary representation of the code value. Thus, valid lengths are within the range of 0 to 32. Note that you are responsible for specifying correct tables with prefix codes for values specified in the value field of the `IppsVLCTable` structure.

VLCEncodeInitAlloc

Allocates memory and initializes the VLC encoder structure.

Syntax

```
ippsVLCEncodeInitAlloc_32s (const IppsVLCTable* pInputTable, int inputTableSize,  
    IppsVLCEncodeSpec_32s** ppVLCSpec);
```

Parameters

<i>pInputTable</i>	Pointer to the VLC table.
<i>inputTableSize</i>	Size of the VLC table.
<i>ppVLCSpec</i>	Double pointer to the VLC encoder specification structure.

Description

The function `ippsVLCEncodeInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the VLC encoder specification structure *ppVLCSpec* that uses the VLC table *pInputTable*. The VLC encoder specification structure is used by the functions [ippsVLCEncodeBlock](#) and [ippsVLCEncodeOne](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.
<code>ippStsVLCInputDataErr</code>	Indicates an error if <i>inputTableSize</i> is less than or equal to 0.

VLCEncodeFree

Frees memory allocated for the VLC encoder structure.

Syntax

```
void ippsVLCEncodeFree_32s (IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

pVLCSpec Pointer to the VLC encoder specification structure.

Description

The function `ippsVLCEncodeFree` is declared in the `ippdc.h` file. This function frees memory allocated for the VLC encode structure by the function [ippsVLCEncodeInitAlloc](#).

VLCEncodeInit

Initializes the VLC encoder structure.

Syntax

```
ippsVLCEncodeInit_32s (const IppsVLCTable_32s* pInputTable, int inputTableSize,
    IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

pInputTable Pointer to the VLC table

inputTableSize Size of the VLC table

pVLCSpec Pointer to the VLC encoder specification structure.

Description

The function `ippsVLCEncodeInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the VLC encoder specification structure *pVLCSpec* that uses the VLC table *pInputTable*. The size of the external buffer should be computed previously by calling the function [ippsVLCEncodeGetSize](#). Alternatively, the VLC encoder structure can be initialized by the function [ippsVLCEncodeInitAlloc](#).

The VLC encoder specification structure is used by the functions [ippsVLC EncodeBlock](#) and [ippsVLC EncodeOne](#).



NOTE. The parameters *pInputTable* and *inputTableSize* of the [ippsVLC EncodeInit](#) and [ippsVLC EncodeGetSize](#) functions must have equal respective values to avoid an error in subsequent encoding functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsVLCInputDataErr</code>	Indicates an error if <i>inputTableSize</i> is less than or equal to 0.

VLC Encode Get Size

Computes the size of the VLC encoder structure.

Syntax

```
ippsVLC EncodeGetSize_32s (const IppsVLC Table_32s* pInputTable, int
    inputTableSize, Ipp32s* pSize);
```

Parameters

<i>pInputTable</i>	Pointer to the VLC table
<i>inputTableSize</i>	Size of the input table
<i>pSize</i>	Pointer to the size of the VLC encoder specification structure.

Description

The function `ippsVLC EncodeGetSize` is declared in the `ippdc.h` file. This function computes the size of the VLC encoder structure that uses the VLC table *pInputTable*. Its size is stored in the *pSize*. This function should be called prior to the function [ippsVLC EncodeInit](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsVLCInputDataErr</code>	Indicates an error if <i>inputTableSize</i> is less than or equal to 0.

VLCEncodeBlock

Performs VLC encoding of a block of data.

Syntax

```
ippVLCEncodeBlock_16slu (const Ipp16s* pSrc, int srcLen, Ipp8u** ppDst, int*
    pDstBitsOffset, const IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Number of elements in the <i>pSrc</i> .
<i>ppDst</i>	Double pointer to the destination bitstream.
<i>pDstBitsOffset</i>	Pointer to the input/output bit position in the <i>pDst</i> .
<i>pVLCSpec</i>	Pointer to the VLC encoder specification structure.

Description

The function `ippVLCEncodeBlock` is declared in the `ippdc.h` file. This function encodes *srcLen* elements of the source vector *pSrc* and stores the result in the destination buffer *pDst*. After encoding the function shifts the *pDst* pointer by the number of successfully written bytes. The function updates *pDstBitsOffset* setting it to the actual bit position in the destination buffer.

The `ippVLCEncodeBlock` function uses the VLC table from the encoder specification structure *pVLCSpec* that must be initialized by the functions [ippVLCEncodeInitAlloc](#) or [ippVLCEncodeInit](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used.

VLCEncodeOne

Performs VLC encoding of a single element.

Syntax

```
ippVLCEncodeBlock_16slu (Ipp16s src, Ipp8u** ppDst, int* pDstBitsOffset, const  
    IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

<code>src</code>	Source value.
<code>ppDst</code>	Double pointer to the destination bitstream.
<code>pDstBitsOffset</code>	Pointer to the input/output bit position in the <code>ppDst</code> .
<code>pVLCSpec</code>	Pointer to the VLC encoder specification structure.

Description

The function `ippVLCDecodeOne` is declared in the `ippdc.h` file. This function encodes a single element `src` and stores the result in the destination buffer `ppDst`. After encoding the function shifts the `ppDst` pointer by the number of successfully written bytes. The function updates `pDstBitsOffset` setting it to the actual bit position in the destination buffer.

The `ippVLCDecodeBlock` function uses the VLC table from the encoder specification structure `pVLCSpec` that must be initialized by the functions [ippVLCDecodeInitAlloc](#) or [ippVLCDecodeInit](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used.

VLCCountBits

Computes a number of bits required to encode the source block.

Syntax

```
ippSVLCCountBits_16s32s (const Ipp16s* pSrc, int srcLen, Ipp32s* pCountBits,
    const IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Number of elements in the <i>pSrc</i> .
<i>pCountBits</i>	Pointer to the computed length in bits required to encode <i>pSrc</i> .
<i>pVLCSpec</i>	Pointer to the VLC encoder specification structure.

Description

The function `ippSVLCCountBits` is declared in the `ippdc.h` file. This function computes a number of bits required to encode the source data *pSrc* using the VLC table from the encoder specification structure *pVLCSpec* that must be initialized by the functions [ippSVLCDecodeInit](#) or [ippSVLCDecodeInitAlloc](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.

VLCDecodeInitAlloc

Allocates memory and initializes the VLC decoder structure.

Syntax

```
ippsVLCDecodeInitAlloc_32s (const IppsVLCTable_32s* pInputTable,  
    int inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables,  
    IppsVLCDecodeSpec_32s** ppVLCSpec);
```

Parameters

<i>pInputTable</i>	Pointer to the VLC table.
<i>inputTableSize</i>	Size of the VLC table.
<i>pSubTablesSizes</i>	Pointer to the sizes of the subtables.
<i>numSubTables</i>	Number of the subtables.
<i>ppVLCSpec</i>	Double pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the VLC decoder structure *ppVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*. Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

The VLC decoder specification structure is used by the functions [ippsVLCDecodeBlock](#) and [ippsVLCDecodeOne](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.

`ippStsVLCUsrTblCodeLengthErr` Indicates an error if the maximum code length in the input table exceeds 32;
or if any size value for the subtables in `pSubTablesSizes` is less than 0;
or if the sum of these values is less than the maximum code length in the input table.

VLCDecodeFree

Frees memory allocated for the VLC decoder structure.

Syntax

```
void ippVLCDecodeFree_32s (IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

`pVLCSpec` Pointer to the VLC decoder specification structure.

Description

The function `ippVLCDecodeFree` is declared in the `ippdc.h` file. This function frees memory allocated for the VLC decoder structure by the function [ippVLCDecodeInitAlloc](#).

VLCDecodeInit

Initializes the VLC decoder structure.

Syntax

```
ippVLCDecodeInit_32s (const IppsVLCTable_32s* pInputTable, int inputTableSize,  
Ipp32s* pSubTablesSizes, int numSubTables, IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

`pInputTable` Pointer to the VLC table.
`inputTableSize` Size of the VLC table.
`pSubTablesSizes` Pointer to the sizes of the subtables.

<i>numSubTables</i>	Number of the subtables.
<i>pVLCSpec</i>	Pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the VLC decoder structure *pVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*. Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

The size of the external buffer should be computed previously by calling the function [ippsVLCDecodeGetSize](#). Alternatively, the VLC decoder structure can be initialized by the function [ippsVLCDecodeInitAlloc](#).

The VLC decoder specification structure is used by the functions [ippsVLCDecodeBlock](#) and [ippsVLCDecodeOne](#).



NOTE. The parameters *pInputTable* and *inputTableSize* of the `ippsVLCDecodeInit` and [ippsVLCDecodeGetSize](#) functions must have equal respective values to avoid an error in subsequent decoding functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsVLCUsrTblCodeLengthErr</code>	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in <i>pSubTablesSizes</i> is less than 1; or if the sum of these values is less than the maximum code length in the input table.

VLCDecodeGetSize

Computes the size of the VLC decoder structure.

Syntax

```
ippVLCDecodeGetSize_32s (const IppsVLCTable_32s* pInputTable, int
    inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables, Ipp32s* pSize);
```

Parameters

<i>pInputTable</i>	Pointer to the input table.
<i>inputTableSize</i>	Size of the input table.
<i>pSubTablesSizes</i>	Pointer to the sizes of the subtables.
<i>numSubTables</i>	Number of the subtables.
<i>pVLCSpec</i>	Pointer to the size of the VLCDecoder specification structure.

Description

The function `ippVLCDecodeGetSize` is declared in the `ippdc.h` file. This function computes the size of the VLC decoder structure *ppVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*. Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

This function should be called prior to the function [ippVLCDecodeInit](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsVLCUsrTblCodeLengthErr</code>	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in <i>pSubTablesSizes</i> is less than 0; or if the sum of these values is less than the maximum code length in the input table.

VLCDecodeBlock

Decodes a block of VLC encoded data.

Syntax

```
ippsVLCDecodeBlock_lu16s (const Ipp8u** ppSrc, int* pSrcBitsOffset, Ipp16s*
    pDst, int dstLen, const IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

<i>ppSrc</i>	Double pointer to the source bitstream.
<i>pSrcBitsOffset</i>	Pointer to the input/output bit position.
<i>pDst</i>	Pointer to the destination buffer.
<i>dstLen</i>	Number of elements in the <i>pDst</i> .
<i>pVLCSpec</i>	Pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeBlock` is declared in the `ippdc.h` file. This function decodes *dstLen* VLC encoded elements in the source bitstream starting from the *pSrcBitsOffset* bit position. The function stores the result in the destination buffer *pDst*. After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates *pSrcBitsOffset* setting it to the actual bit position in the source buffer.

The function `ippsVLCDecodeBlock` uses the VLC table from the decoder specification structure *pVLCSpec* that must be initialized by the functions [ippsVLCDecodeInitAlloc](#) or [ippsVLCDecodeInit](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used. It can indicate that bitstream contains code that is not specified in the table.
<code>ippStsBitOffsetErr</code>	Indicates an error if value of the <i>pSrcBitsOffset</i> is less than 0 or greater than 7.

<code>ippStsContextMatchErr</code>	Indicates an error if the <i>pVLCSpec</i> structure is not initialized by the function ippsVLCDecodeInitAlloc or ippsVLCDecodeInit .
------------------------------------	--

VLCDecodeOne

Decodes a single VLC encoded element.

Syntax

```
ippsVLCDecodeOne_lu16s (const Ipp8u** ppSrc, int* pSrcBitsOffset, Ipp16s* pDst,
    const IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

<i>ppSrc</i>	Double pointer to the source bitstream.
<i>pSrcBitsOffset</i>	Pointer to the input/output bit position.
<i>pDst</i>	Pointer to the decoded value.
<i>pVLCSpec</i>	Pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeOne` is declared in the `ippdc.h` file. This function decodes a single VLC encoded element in the *ppSrc* on position *pSrcBitsOffset*. The decoded element is stored in the destination buffer *pDst*.

After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates *pSrcBitsOffset* setting it to the actual bit position in the source buffer.

The function `ippsVLCDecodeOne` uses the VLC table from the decoder specification structure *pVLCSpec* that must be initialized by the functions [ippsVLCDecodeInitAlloc](#) or [ippsVLCDecodeInit](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.

<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used. For decoding functions it can indicate that bitstream contains code that is not specified in the table.
<code>ippStsBitOffsetErr</code>	Indicates an error if value of the <code>pSrcBitsOffset</code> is less than 0 or greater than 7.
<code>ippStsContextMatchErr</code>	Indicates an error if the <code>pVLCSpec</code> structure is not initialized by the function ippsVLCDecodeInitAlloc or ippsVLCDecodeInit .

Example of Using VLC Coding Functions

The functions `ippsVLCCountBits` and `ippsVLCEncodeBlock` can be simply used in the wide range of different encoders (for example, for audio data). For example, if you need to quantize some vectors and encode them using no more bits than specified, these functions can be used as follows:

- previously the table `inputTable` of the size `inputTableSize` should be created; this table is of the following format:

```
static IppsVLCTable_32s inputTable[] =
{
    {value0, code0, length0};
    {value1, code1, length1};
    ...
    {valueN, codeN, lengthN};
}
```

- initialization step:

```
ippsVLCEncodeGetSize_32s(InputTable, inputTableSize, &Size);
pVLCSpec = ippsMalloc_8u(Size); /* memory allocation */
ippsVLCEncodeInit_32s(pInputTable, inputTableSize, pVLCSpec);
...
```

- counting bits step:

```
Do {
    //A some quantization algorithm should be here. Let pSrc is output of
    this algorithm.
    ...
    ippsVLCCountBits_16s32s(pSrc, srcLen, &bitNumber, pVLCSpec);
} while (bitNumber > allowedBitNumber);
```

- encoding bit stream step:

```
ippsVLCEncodeBlock_16slu (pSrc, srcLen, ppBitStream,
pBitStreamBitsOffset, pVLCSpec);
```


Huffman Coding

This section describes functions implementing the Huffman coding algorithm. This algorithm performs so called entropy encoding. The main idea of this method is to substitute the code words (Huffman codes) instead of symbols. The least length codes correspond to the most often occurring symbols allowing to perform compression. For example, the sequence “aaaaaaaa” contains only one symbol ‘a’ that corresponds to the Huffman code with size 1 bit; after encoding the output sequence will have size 1 byte - thus compression ratio will be 8.

Limitations and conventions.

Intel IPP functions implement the Huffman static encoding. This means that encoding is a two-passes operation. In the first pass the probability structure of the source data is determined, then in the second pass the encoding itself is performed. The table of Huffman Code lengths must be passed to decoder (this table can be effectively compressed).

Maximum length of the Huffman codes in the Intel IPP functions for Huffman coding is 32 bits.

The functions for Huffman encoding/decoding initialize a correspondent specification structure and perform the encoding and decoding operations.

To use the functions for **encoding**, follow this general scheme:

- Call the function [ippsEncodeHuffInitAlloc](#) to allocate memory and initialize the Huffman structure for encoding with specified frequency table.
Or call the function [ippsEncodeHuffInit](#) to initialize the Huffman structure for encoding in the previously created external buffer. The size of this buffer can be computed by calling the function [ippsHuffGetSize](#).
- If encoding is performed **by blocks**, call the function [ippsHuffGetDstBuffSize](#) to compute the size of the destination buffer.
- Call [ippsEncodeHuffOne](#) to encode a single element, or call [ippsEncodeHuff](#) to encode the source sequence (vector). (Alternatively the sequence of N element can be encoded by calling [ippsEncodeHuffOne](#) N times.) Then call [ippsEncodeHuffFinal](#) to finalize the encoding procedure. Note that the size of the input sequence (vector) should be passed to the decoding functions.
- Call [ippsHuffGetLenCodeTable](#) to received the table containing the lengths of Huffman codes. This table should be passed to the decoding functions. To increase the effectiveness of encoding it may be compressed by calling the function [ippsHuffLenCodeTablePack](#).
- Call [ippsHuffFree](#) to free dynamic memory allocated with the Huffman structure for encoding by the function [ippsEncodeHuffInitAlloc](#).

To use these functions for **decoding**, follow this general scheme:

- Call the function [ippsDecodeHuffInitAlloc](#) to allocate memory and initialize the Huffman structure for decoding with specified table containing lengths of Huffman codes. Or call the function [ippsDecodeHuffInit](#) to initialize the Huffman structure for encoding in the previously created external buffer. The size of this buffer can be computed by calling the function [ippsHuffGetSize](#).
- If decoding is performed **by blocks**, call the function [ippsHuffGetDstBuffSize](#) to compute the size of the destination buffer.
- Call [ippsDecodeHuffOne](#) to encode a single element, or call [ippsDecodeHuff](#) to encode the source sequence (vector). The size of the source non-encoded sequence (vector) should be passed from the encoding functions as well as the table containing the lengths of Huffman codes. If this table is in the packed form, it may be restored by calling the function [ippsHuffLenCodeTableUnpack](#).
- Call [ippsHuffFree](#) to free dynamic memory allocated with the Huffman structure for decoding by the function [ippsDecodeHuffInitAlloc](#).

EncodeHuffInitAlloc

Allocates memory and initializes the structure for Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuffInitAlloc_8u(const int freqTable[256],  
    IppHuffState_8u** ppHuffState);
```

Parameters

<i>freqTable</i>	Table of frequencies of the symbols.
<i>ppHuffState</i>	Double pointer to the Huffman encoding structure.

Description

The function `ippsEncodeHuffInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the structure for Huffman encoding *pHuffState*. This structure is specifying by the table of symbol's frequencies *freqTable*. The Huffman encoding structure is used by the functions [ippsEncodeHuffOne](#) and [ippsEncodeHuff](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pHuffState</i> pointer is NULL.
<code>ippStsFreqTableErr</code>	Indicates an error if <i>freqTable</i> is invalid.
<code>ippStsMaxLenHuffCodeErr</code>	Indicates an error if the length of the Huffman code is more than expected one.
<code>ippMemAllocErr</code>	Indicates an error if memory allocation fails.

HuffFree

Frees memory allocated for the Huffman encoding and decoding structures.

Syntax

```
void ippHuffFree_8u(IppHuffState_8u* pHuffState);
```

Parameters

pHuffState Pointer to the Huffman coding structure.

Description

The function `ippHuffFree` is declared in the `ippdc.h` file. This function frees memory allocated for the Huffman encoding and decoding structures by the function [`ippEncodeHuffInitAlloc`](#) and [`ippDecodeHuffInitAlloc`](#) respectively.

EncodeHuffInit

Initializes the Huffman encoding structure.

Syntax

```
IppStatus ippEncodeHuffInit_8u (const int freqTable[256], IppHuffState_8u*
    pHuffState);
```

Parameters

<i>freqTable</i>	Table of frequencies of the symbols.
<i>pHuffState</i>	Pointer to the Huffman encoding structure.

Description

The function `ippsEncodeHuffInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the structure *pHuffState* that is required for Huffman encoding. This structure is specifying by the table of symbol's frequencies *freqTable*. The size of the external buffer should be computed previously by calling the function [ippsHuffGetSize](#). Alternatively, the Huffman encoder structure can be initialized by the function [ippsEncodeHuffInitAlloc](#).

The Huffman encoding structure is used by the functions [ippsEncodeHuffOne](#) and [ippsEncodeHuff](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pHuffState</i> pointer is NULL.
<code>ippStsFreqTableErr</code>	Indicates an error if <i>freqTable</i> is invalid.
<code>ippStsMaxLenHuffCodeErr</code>	Indicates an error if the length of the Huffman code is more than expected one.

HuffGetSize

Computes size of the external buffer for the Huffman encoding structure.

Syntax

```
IppStatus ippsHuffGetSize_8u(int* pHuffStateSize);
```

Parameters

<i>pHuffStateSize</i>	Pointer to the computed size of the Huffman coding structure.
-----------------------	---

Description

The function `ippsHuffGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) for the external buffer required for the Huffman encoding structure. This function should be called prior to the function [ippsEncodeHuffInit](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <code>pHuffStateSize</code> pointer is NULL.

EncodeHuffOne

Performs Huffman encoding of a single element.

Syntax

```
IppStatus ippsEncodeHuffOne_8u(Ipp8u src, Ipp8u* pDst, int dstOffsetBits,  
    IppHuffState_8u* pHuffState);
```

Parameters

<code>src</code>	Source single element.
<code>pDst</code>	Pointer to the destination buffer.
<code>dstOffsetBits</code>	Offset (in bits) of the position of code word in the destination buffer, starting from the high bit.
<code>pHuffState</code>	Pointer to the Huffman encoding structure.

Description

The function `ippsEncodeHuffOne` is declared in the `ippdc.h` file. This function encodes the single source element `src` using the Huffman encoding structure `pHuffState`. The result is placed in the destination buffer `pDst` in position specified by the value (in bits) of offset from the high bit `dstOffsetBits`. The Huffman encoding structure should be initialized by calling the functions [ippsEncodeHuffInit](#) or [ippsEncodeHuffInitAlloc](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.

EncodeHuff

Performs Huffman encoding.

Syntax

```
IppStatus ippEncodeHuff_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int*  
    pDstLen, IppHuffState_8u* pHuffState);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Number of elements in the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the number of elements in the destination vector after encoding.
<i>pHuffState</i>	Pointer to the Huffman encoding structure.

Description

The function `ippEncodeHuff` is declared in the `ippdc.h` file. This function encodes data in the source buffer *pSrc* using the Huffman encoding structure *pHuffState*. The encoded data are placed in the destination buffer *pDst*. The actual number of elements in the destination buffer is written in *pDstLen*. The Huffman encoding structure should be initialized by calling the functions [ippEncodeHuffInit](#) or [ippEncodeHuffInitAlloc](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>srcLen</i> is less than or equal to 0.

EncodeHuffFinal

Performs Huffman encoding of the remainder.

Syntax

```
IppStatus ippsEncodeHuffFinal_8u(Ipp8u* pDst, int* pDstLen, IppHuffState_8u* pHuffState);
```

Parameters

<i>pDst</i>	Pointer to the destination vector.
<i>pDstLen</i>	Pointer to the number of elements in the destination vector after encoding.
<i>pHuffState</i>	Pointer to the the Huffman coding structure.

Description

The function `ippsEncodeHuffFinal` is declared in the `ippdc.h` file. This function performs the final stage of the Huffman encoding - flushes the remainder part of the encoded data to the destination vector *pDst* with *pDstLen* elements using the encoding structure *pHuffState*. The function should be run just after [ippsEncodeHuff](#), while the Huffman encoding structure *pHuffState* is still initialized.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.

HuffGetLenCodeTable

Returns the table containing lengths of Huffman codes.

Syntax

```
IppStatus ippsHuffGetLenCodeTable_8u(int codeLenTable[256], IppHuffState_8u* pHuffState);
```

Parameters

<i>codeLenTable</i>	Destination table containing lengths of Huffman codes.
<i>pHuffState</i>	Pointer to the pointer to the Huffman decoding structure.

Description

The function `ippsHuffGetLenCodeTable` is declared in the `ippdc.h` file. This function creates the table *codeLenTable* containing the lengths of the used Huffman codes. Their value are taken from the Huffman encoding structure *pHuffState*, which must be initialized beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pHuffState</i> pointer is NULL.

DecodeHuffInitAlloc

Allocates memory and initializes the Huffman decoding structure.

Syntax

```
IppStatus ippsDecodeHuffInitAlloc_8u(const int codeLenTable[256],  
    IppHuffState_8u** ppHuffState);
```

Parameters

<i>codeLenTable</i>	Table containing lengths of Huffman codes.
<i>ppHuffState</i>	Double pointer to the Huffman decoding structure.

Description

The function `ippsDecodeHuffInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the Huffman decoding structure *pHuffState*. The structure requires the table of lengths of Huffman codes *codeLenTable* that was used in the encoding operation. This table can be obtained by calling the function [ippsHuffGetLenCodeTable](#) or [ippsHuffLenCodeTableUnpack](#). The Huffman decoding structure is used by the functions [ippsDecodeHuffOne](#) and [ippsDecodeHuff](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <code>pHuffState</code> pointer is NULL.
<code>ippMemAllocErr</code>	Indicates an error if memory allocation fails.

DecodeHuffInit

Initialized the Huffman decoding structure.

Syntax

```
IppStatus ippDecodeHuffInit_8u(const int codeLenTable[256], IppHuffState_8u*
    pHuffState);
```

Parameters

<code>codeLenTable</code>	Table containing lengths of Huffman codes.
<code>pHuffState</code>	Pointer to the Huffman decoding structure.

Description

The function `ippDecodeHuffInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the Huffman decoding structure `pHuffState`. The structure requires the table of lengths of Huffman codes `codeLenTable` that was used in the encoding operation. This table can be obtained by calling the function [ippHuffGetLenCodeTable](#) or [ippHuffLenCodeTableUnpack](#). The size of the external buffer should be computed previously by calling the function [ippHuffGetSize](#). Alternatively, the Huffman decoding structure can be initialized by the function [ippDecodeHuffInitAlloc](#).

The Huffman decoding structure is used by the functions [ippDecodeHuffOne](#) and [ippDecodeHuff](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <code>pHuffState</code> pointer is NULL.

DecodeHuffOne

Decodes a single code word.

Syntax

```
IppStatus ippsDecodeHuffOne_8u(const Ipp8u* pSrc, int srcOffsetBits, Ipp8u* pDst,  
    IppHuffState_8u* pHuffState);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcOffsetBits</i>	Offset (bits) in the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pHuffState</i>	Pointer to the Huffman decoding structure.

Description

The function `ippsDecodeHuffOne` is declared in the `ippdc.h` file. This function performs Huffman decoding of the single code word in the source buffer *pSrc*. Its position is specified by the value (in bits) of offset from the high bit *srcOffsetBits*. The decoded element is stored in the destination buffer *pDst*. The function uses the Huffman decoding structure *pHuffState* that should be initialized by calling the functions [ippsDecodeHuffInitAlloc](#) or [ippsDecodeHuffInit](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.

DecodeHuff

Performs Huffman decoding.

Syntax

```
IppStatus ippsDecodeHuff_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int*  
    pDstLen, IppHuffState_8u* pHuffState);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Number of elements in the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the expected number of elements in the destination buffer; after decoding to the actual number of elements in the destination buffer.
<i>pHuffState</i>	Pointer to the Huffman coding structure.

Description

The function `ippsDecodeHuff` is declared in the `ippdc.h` file. This function decodes the data in the source buffer *pSrc* containing *srcLen* code words. The *pDstLen* decoded elements are stored in the destination vector *pDst*. The function uses the Huffman decoding structure *pHuffState* that must be initialized by calling the functions [ippsDecodeHuffInitAlloc](#) or [ippsDecodeHuffInit](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>srcLen</i> is less than or equal to 0.

HuffGetDstBuffSize

Computes the size of a destination buffer for Huffman encoding/decoding by blocks.

Syntax

```
IppStatus ippsHuffGetDstBuffSize_8u(const int codeLenTable[256], int srcLen,
    int* pEncDstBuffSize, int* pDecDstBuffSize);
```

Parameters

<i>codeLenTable</i>	Table with lengths of Huffman codes.
<i>srcLen</i>	Number of elements in the source buffer.

<i>pEncDstBuffSize</i>	Pointer to the computed size of the destination buffer for Huffman encoding.
<i>pDecDstBuffSize</i>	Pointer to the computed size of the destination buffer for Huffman decoding.

Description

The function `ippsHuffGetDstBuffSize` is declared in the `ippdc.h` file. This function computes the size (in bytes) of destination buffer for Huffman encoding and decoding by blocks. It should be called prior to the encoding/decoding operations.

The function computes the size of destination buffers for both operations. To exclude one of them the correspondent pointer should be set to `NULL`.

Application Notes

The following formula can be used to estimate maximum sizes of the destination buffer for the Huffman encoding and decoding by blocks:

```
encDstBuffSize = srcLen*4,  
decDstBuffSize = srcLen*8
```

You can use these values without calling the function `ippsHuffGetDstBuffSize`. Nevertheless the function `ippsHuffGetDstBuffSize` allows to minimize the sizes of the destination buffers.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>codeLenTable</i> pointer is <code>NULL</code> .

HuffLenCodeTablePack

Packs the table containing lengths of Huffman codes.

Syntax

```
IppStatus ippsHuffLenCodeTablePack_8u(const int codeLenTable[256], Ipp8u* pDst,  
int* pDstLen);
```

Parameters

<i>codeLenTable</i>	Table containing lengths of Huffman codes.
---------------------	--

<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the number of elements in the destination buffer; after operation to the actual number of elements in the destination buffer.

Description

The function `ippsHuffLenCodeTablePack` is declared in the `ippdc.h` file. This function compresses the table `codeLenTable` containing lengths of Huffman codes and that is required for decoding. The table may be retrieved by calling the function [ippsHuffGetLenCodeTable](#). To restore the table from the compressed form, call the function [ippsHuffLenCodeTableUnpack](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>pDstLen</i> is less than or equal to 0; or if <i>pDstLen</i> is less than actual size of the output.

HuffLenCodeTableUnpack

Unpacks the table containing lengths of Huffman codes.

Syntax

```
IppStatus ippsHuffLenCodeTableUnpack_8u(const Ipp8u* pSrc, int* pSrcLen, int
codeLenTable[256]);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the number of elements in the source buffer; after operation pointer to the actual number of elements in the source buffer.
<i>codeLenTable</i>	Table containing lengths of Huffman codes.

Description

The function `ippsHuffLenCodeTableUnpack` is declared in the `ippdc.h` file. This function restores the table `codeLenTable` containing the lengths of Huffman codes from the packed form `pSrc` created by the function [ippsHuffLenCodeTablePack](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>pSrcLen</code> is less than or equal to 0; or if <code>pSrcLen</code> is less than actual size.

Dictionary-Based Compression Functions

This section describes the Intel IPP functions that use different dictionary-based compression methods.

The full list of these functions is given in the [Table 13-3](#)

Table 13-3 Intel IPP Dictionary-Based Data Compression Functions

Function Short Name	Description
LZSS Coding	
EncodeLZSSInitAlloc	Allocates memory and initializes the LZSS state structure.
LZSSFree	Frees memory allocated for the LZSS state structure.
EncodeLZSSInit	Initializes the LZSS state structure.
LZSSGetSize	Computes the size of the LZSS state structure.
EncodeLZSS	Performs LZSS encoding.
EncodeLZSSFlush	Flushes the last few bits in the bitstream and aligns the output data on a byte boundary.
DecodeLZSSInitAlloc	Allocates memory and initializes the LZSS state structure.
DecodeLZSSInit	Initializes the LZSS state structure.
DecodeLZSS	Performs LZSS decoding
ZLIB Coding	
EncodeLZ77Init	Initializes the encoding LZ77 state structure.
EncodeLZ77GetSize	Computes the size of the encoding LZ77 state structure.
EncodeLZ77InitAlloc	Allocates memory and initializes the encoding LZ77 state structure.
LZ77Free	Frees memory allocated for the encoding or decoding LZ77 state structures.
EncodeLZ77	Performs LZ77 encoding.
EncodeLZ77SelectHuffMode	Determines the optimal encoding mode.
EncodeLZ77FixedHuff	Performs encoding with the fixed Huffman codes.
EncodeLZ77DynamicHuff	Performs encoding with the dynamic Huffman codes.
EncodeLZ77StoredBlock	Stores the data block without compression.
EncodeLZ77Flush	Writes the checksum and total length of the input data to the end of the output stream.
EncodeLZ77GetPairs	Retrieves pair data from the encoding state structure.
EncodeLZ77SetPairs	Sets pair data to the encoding state structure.

Table 13-3 Intel IPP Dictionary-Based Data Compression Functions (continued)

Function Short Name	Description
<u>EncodeLZ77GetStatus</u>	Reads the deflate status value from the encoding state structure.
<u>EncodeLZ77SetStatus</u>	Writes the deflate status value to the encoding state structure.
<u>EncodeLZ77Reset</u>	Resets the internal encoding state structure to the initial state.
<u>DecodeLZ77Init</u>	Initializes the LZ77 decoding structure.
<u>DecodeLZ77GetSize</u>	Computes the size of the decoding LZ77 state structure.
<u>DecodeLZ77InitAlloc</u>	Allocates memory and initializes the decoding LZ77 state structure.
<u>DecodeLZ77</u>	Performs LZ77 decoding.
<u>DecodeLZ77GetBlockType</u>	Determines the type of encoded data block.
<u>DecodeLZ77FixedHuff</u>	Performs fixed Huffman decoding.
<u>DecodeLZ77DynamicHuff</u>	Performs dynamic Huffman decoding.
<u>DecodeLZ77StoredBlock</u>	Performs decoding of the block stored without compression.
<u>DecodeLZ77GetPairs</u>	Retrieves pair data from the decoding state structure.
<u>DecodeLZ77SetPairs</u>	Sets pair data to the decoding state structure.
<u>DecodeLZ77GetStatus</u>	Reads the inflate status value from the LZ77 decoding state structure.
<u>DecodeLZ77SetStatus</u>	Sets the inflate status to the desired value in the LZ77 decoding state structure
<u>DecodeLZ77Reset</u>	Resets the internal decoding state structure to the initial state.
<u>Adler32</u>	Computes the Adler32 checksum for the source data buffer.
<u>CRC32</u>	Computes the CRC32 checksum for the source data buffer.

LZSS Compression Functions

This section describes Intel IPP data compression functions that implement the LZSS (Lempel-Ziv-Storer-Szymanski) compression algorithm [[Storer82](#)]. These functions perform LZSS coding with a vocabulary size of 32KB and 256-byte maximum match string length.

EncodeLZSSInitAlloc

Allocates memory and initializes the LZSS encoder state structure.

Syntax

```
IppStatus ippsEncodeLZSSInitAlloc_8u (IppLZSSState_8u** ppLZSSState);
```

Parameters

ppLZSSState Double pointer to the LZSS encoder state structure.

Description

The function `ippsEncodeLZSSInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the LZSS encoder state structure *ppLZSSState* required for the encoder functions [ippsEncodeLZSS](#) and [ippsEncodeLZSSFlush](#).

Return Values

<code>ippStsNoErr</code>	Indicates no errors.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>ppLZSSState</i> pointer is NULL.
<code>StsMemAllocErr</code>	Indicates an error if memory allocation fails.

LZSSFree

Frees memory allocated for the LZSS state structure.

Syntax

```
void ippsLZSSFree_8u (IppLZSSState_8u* pLZSSState);
```

Parameters

pLZSSState Pointer to the LZSS encoder state structure.

Description

The function `ippsEncodeLZSSFree` is declared in the `ippdc.h` file. This function frees memory allocated for the LZSS state structure for encoding and decoding by the function [ippsEncodeLZSSInitAlloc](#) or [ippsDecodeLZSSInitAlloc](#) respectively.

EncodeLZSSInit

Initializes the LZSS encoder state structure.

Syntax

```
IppStatus ippsEncodeLZSSInit_8u (IppLZSSState_8u* pLZSSState);
```

Parameters

pLZSSState Pointer to the LZSS encoder state structure.

Description

The function `ippsEncodeLZSSInit` is declared in the `ippdc.h` file. This function initializes the LZSS state structure *pLZSSState* in the external buffer. Its size should be computed previously by calling the function [ippsLZSSGetSize](#). Alternatively, this state structure can be initialized by the function [ippsEncodeLZSSInitAlloc](#).

The LZSS encoder state structure is required for the encoder functions [ippsEncodeLZSS](#) and [ippsEncodeLZSSFlush](#).

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error if the pointer *pLZSSState* is NULL.

LZSSGetSize

Computes the size of the LZSS state structure.

Syntax

```
IppStatus ippsLZSSGetSize_8u (int* pLZSSStateSize);
```

Parameters

pLZSSStateSize Pointer to the size of the LZSS state structure.

Description

The function `ippsLZSSGetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the LZSS state structure for encoding and decoding and stores it to an integer pointed to by *pLZSSStateSize*. The function should be called prior to the function [ippsEncodeLZSSInit](#) or [ippsDecodeLZSSInit](#).

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error if the pointer *pLZSSStateSize* is NULL.

EncodeLZSS

Performs LZSS encoding.

Syntax

```
IppStatus ippsEncodeLZSS_8u (Ipp8u** ppSrc, int* pSrcLen, Ipp8u** ppDst,
    int* pDstLen, IppLZSSState_8u* pLZSSState);
```

Parameters

ppSrc Double pointer to the source buffer.

pSrcLen Pointer to the number of elements in the source buffer; it is updated after encoding.

<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>pLZSSState</i>	Pointer to the LZSS encoding state structure.

Description

The function `ippsEncodeLZSS` is declared in the `ippdc.h` file. This function performs LZSS encoding of data in the source buffer *ppSrc* with *pSrcLen* length and stores the result in the destination vector *pDst*. The length of the destination vector is stored in *pDstLen*. The LZSS encoder state structure *pLZSSState* must be initialized by the functions [ippsEncodeLZSSInitAlloc](#) or [ippsEncodeLZ77Init](#) beforehand.

After encoding the function returns the pointers to source and destination buffers shifted by the number of successfully read and encoded bytes, respectively. The function updates *pSrcLen* so it is equal to the actual number of elements in the source buffer.

[Example 13-1](#) shows how to use the function `ippsEncodeLZSS_8u` and supporting functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>srcLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

EncodeLZSSFlush

Encodes the last few bits in the bitstream and aligns the output data on the byte boundary.

Syntax

```
IppStatus ippsEncodeLZSSFlush_8u (Ipp8u** ppDst, int* pDstLen, IppLZSSState_8u* pLZSSState);
```

Parameters

<i>ppDst</i>	Double pointer to the destination buffer.
--------------	---

<i>pDstLen</i>	Pointer to the length of destination buffer.
<i>pLZSSState</i>	Pointer to the LZSS encoder state structure.

Description

The function `ippEncodeLZSSFlush` is declared in the `ippdc.h` file. This function encodes the last few bits (remainder) in the bitstream, writes them to *ppDst*, and aligns the output data on a byte boundary.

[Example 13-1](#) shows how to use the function `ippEncodeLZSSFlush_8u` and supporting functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>pDstLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

Example 13-1 Using the Function `ippEncodeLZSS_8u` and Supporting Functions

```
#define INBLOCKSIZE 65536
#define OUTBLOCKSIZE 16384
.....
#include <stdio.h>
#include "ippdc.h"
.....
FILE          *inFile, *outFile;
Ipp8u         *pSrc = NULL, *pDst = NULL, *startpSrc, *startpDst;
int           srcLen, dstLen = 0;
int           commonDstLen;
IppLZSSState_8u *pLZSSState;
IppStatus     status;
.....

/*****
/* Memory allocation for input data and output code buffers, */
/* and for the internal encoding state structure:                */
*****/
```

Example 13-1 Using the Function `ippsEncodeLZSS_8u` and Supporting Functions

```

startpSrc = ippsMalloc_8u( (INBLOCKSIZE) * sizeof(char) );
startpDst = ippsMalloc_8u( (OUTBLOCKSIZE) * sizeof(char) );
ippsEncodeLZSSInitAlloc_8u(&pLZSSState);

/*****
/* Initializing the arguments, reading srcLen bytes of          */
/* input data from input data file to startpSrc:                */
*****/
commonDstLen = (OUTBLOCKSIZE);
pDst = startpDst;
dstLen = commonDstLen;
srcLen = fread( startpSrc, sizeof(Ipp8u), INBLOCKSIZE, inFile );
if(srcLen <= 0)
    return -1;
pSrc = startpSrc;

/***** */
/* The main loop. In every iteration program calls the          */
/* ippsEncodeLZSS_8u. ippsEncodeLZSS_8u changes the values      */
/* of pSrc, srcLen, pDst and dstLen. ippsEncodeLZSS_8u          */
/* returns ippsStsDstSizeLessExpected if there is no            */
/* room in the destination buffer pDst to continue encoding.    */
/* In this case dstLen == 0. Such case is handled by             */
/* flushing the pDst to the output file. If ippsEncodeLZSS_8u   */
/* returns ippsStsNoErr, then the program flushes the current    */
/* pDst of length of (commonDstLen - dstLen) bytes to           */
/* the output file and reads next srcLen bytes of input data    */
/* to startpSrc.                                                 */
*****/
for( ; ; )
{
    status = ippsEncodeLZSS_8u(&pSrc, &srcLen, &pDst, &dstLen, pLZSSState);
    if( status == ippsStsDstSizeLessExpected )
    {
        fwrite( startpDst, sizeof(Ipp8u), (commonDstLen - dstLen), outFile );
        dstLen = commonDstLen;
        pDst = startpDst;
    }
    else if( status == ippsStsNoErr )
    {
        fwrite( startpDst, sizeof(Ipp8u), (commonDstLen - dstLen), outFile );
        pDst = startpDst;
        dstLen = commonDstLen;
    }
}

```

Example 13-1 Using the Function `ippsEncodeLZSS_8u` and Supporting Functions

```

        srcLen = fread( startpSrc, sizeof(Ipp8u), INBLOCKSIZE, inFile );
        if( srcLen <= 0 )
            break;
        pSrc = startpSrc;
    }
    else
        return (-1);
} /* for */

/*****
/*  The call of ippsEncodeLZSSFlush_8u flushes the last few
/*  bits of code to the destination buffer:
*****/
if( ippsEncodeLZSSFlush_8u(&pDst, &dstLen, pLZSSState) == ippsStsNoErr )
{
    fwrite( startpDst, sizeof(Ipp8u), (commonDstLen - dstLen), outFile );
    ippsLZSSFree_8u(pLZSSState);
    ippsFree( startpSrc );
    ippsFree( startpDst );
    _fcloseall();
}
else
    return (-1);
.....

```

DecodeLZSSInitAlloc

Allocates memory and initializes the LZSS decoder state structure.

Syntax

```
IppStatus ippsDecodeLZSSInitAlloc_8u (IppLZSSState_8u** ppLZSSState)
```

Parameters

<i>ppLZSSState</i>	Double pointer to the LZSS decoder state structure.
--------------------	---

Description

The function `ippsDecodeLZSSInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the LZSS decoder state structure. That structure is used by the function [ippsDecodeLZSS](#).

Return Values

<code>ippStsNoErr</code>	Indicates no errors.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pLZSSState</code> is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.

DecodeLZSSInit

Initializes the LZSS decoder state structure.

Syntax

```
IppStatus ippsDecodeLZSSInit_8u (IppLZSSState_8u* pLZSSState);
```

Parameters

<code>pLZSSState</code>	Pointer to the LZSS decoder state structure.
-------------------------	--

Description

The function `ippsDecodeLZSSInit` is declared in the `ippdc.h` file. This function initializes the LZSS decoder state structure in the external buffer, the size of which should be computed previously by calling the function [ippsLZSSGetSize](#). Alternatively, this state structure can be initialized by the function [ippsDecodeLZSSInitAlloc](#).

The LZSS decoder state structure is required for the function [ippsDecodeLZSS](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pLZSSState</code> is NULL.

DecodeLZSS

Performs LZSS decoding.

Syntax

```
IppStatus ippsDecodeLZSS_8u (Ipp8u* ppSrc, int* pSrcLen, Ipp8u*8 ppDst,
    int* pDstLen, IppLZSSState_8u* pLZSSState);
```

Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>pLZSSState</i>	Pointer to the LZSS decoding state structure.

Description

The function `ippsDecodeLZSS` is declared in the `ippdc.h` file. This function performs LZSS decoding of the *pSrcLen* elements of the *ppSrc* source buffer and stores the result in the *ppDst* destination vector. The length of the destination vector is stored in *pDstLen*. The LZSS decoder state structure *pLZSSState* must be initialized by the functions [ippsDecodeLZSSInitAlloc](#) or [ippsDecodeLZSSInit](#) beforehand.

After decoding the function returns the pointers to source and destination buffers shifted by the number of successfully read and decoded bytes respectively. The function updates *pSrcLen* so it is equal to the actual number of elements in the source buffer.

[Example 13-2](#) shows how to use the `ippsDecodeLZSS_8u` function and supporting functions.

Example 13-2 Using the Function `ippsDecodeLZSS` and Supporting functions

```
#define INBLOCKSIZE 65536
#define OUTBLOCKSIZE 16384
.....
#include <stdio.h>
#include "ippdc.h"
.....
FILE                *inFile, *outFile;
Ipp8u               *pSrc = NULL, *pDst = NULL, *startpSrc, *startpDst;
int                 srcLen, dstLen = 0;
IppLZSSState_8u     *pLZSSState;
IppStatus            status;
.....

/*****
/*  Memory allocation for input code and output restored data */
/*  buffers, and for the internal decoding state structure:  */
*****/
startpSrc = ippsMalloc_8u( OUTBLOCKSIZE * sizeof(Ipp8u) );
startpDst = ippsMalloc_8u( INBLOCKSIZE * sizeof(Ipp8u) );
ippsDecodeLZSSInitAlloc_8u( &pLZSSState );
/*****
/*  Initializing the arguments, reading srcLen bytes of      */
/*  input code from the input code file to startpSrc:        */
*****/
dstLen = INBLOCKSIZE;
srcLen = fread( startpSrc, sizeof(Ipp8u), OUTBLOCKSIZE, inFile );
```

Example 13-2 Using the Function `ippsDecodeLZSS` and Supporting functions

```
pSrc = startpSrc;
pDst = startpDst;

/*****
/* The main loop. In every iteration program calls the ippsDecodeLZSS_8u. */
/* ippsDecodeLZSS_8u changes the values of pSrc, srcLen, pDst and dstLen. */
/* ippsDecodeLZSS_8u* returns ippsStsDstSizeLessExpected if there is no */
/* room in the destination buffer pDst to continue decoding. In this case */
/* dstLen == 0. Such case is handled by flushing the pDst to the output */
/* file. If ippsDecodeLZSS_8u returns ippsStsNoErr, then program flushes */
/* the current pDst of length of (INBLOCKSIZE - dstLen) bytes to the */
/* output file and reads next srcLen bytes of input code to startSrc */
*****/
for( ; ; )
{
    status = ippsDecodeLZSS_8u( &pSrc, &srcLen, &pDst, &dstLen, pLZSSState );
    if( status == ippsStsNoErr )
    {
        fwrite(startpDst, sizeof(Ipp8u), (INBLOCKSIZE - dstLen), outFile);
        pDst = startpDst;
        dstLen = INBLOCKSIZE;
        srcLen = fread( startpSrc, sizeof(Ipp8u), OUTBLOCKSIZE, inFile );
        pSrc = startpSrc;
        if(srcLen == 0)
            break;
    }
    else if( status == ippsStsDstSizeLessExpected )
    {
        fwrite(startpDst, sizeof(Ipp8u), (INBLOCKSIZE - dstLen), outFile);
        pDst = startpDst;
        dstLen = INBLOCKSIZE;
    }
} /* for */
/*****
/* Free the memory allocated for input code and output */
/* restored data buffers and for the internal decoding state */
/* structure: */
*****/
ippsLZSSFree_8u( pLZSSState );
ippsFree( startpSrc );
ippsFree( startpDst );
_fcloseall();
.....
```

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>pSrcLen</i> or <i>pDstLen</i> is negative.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

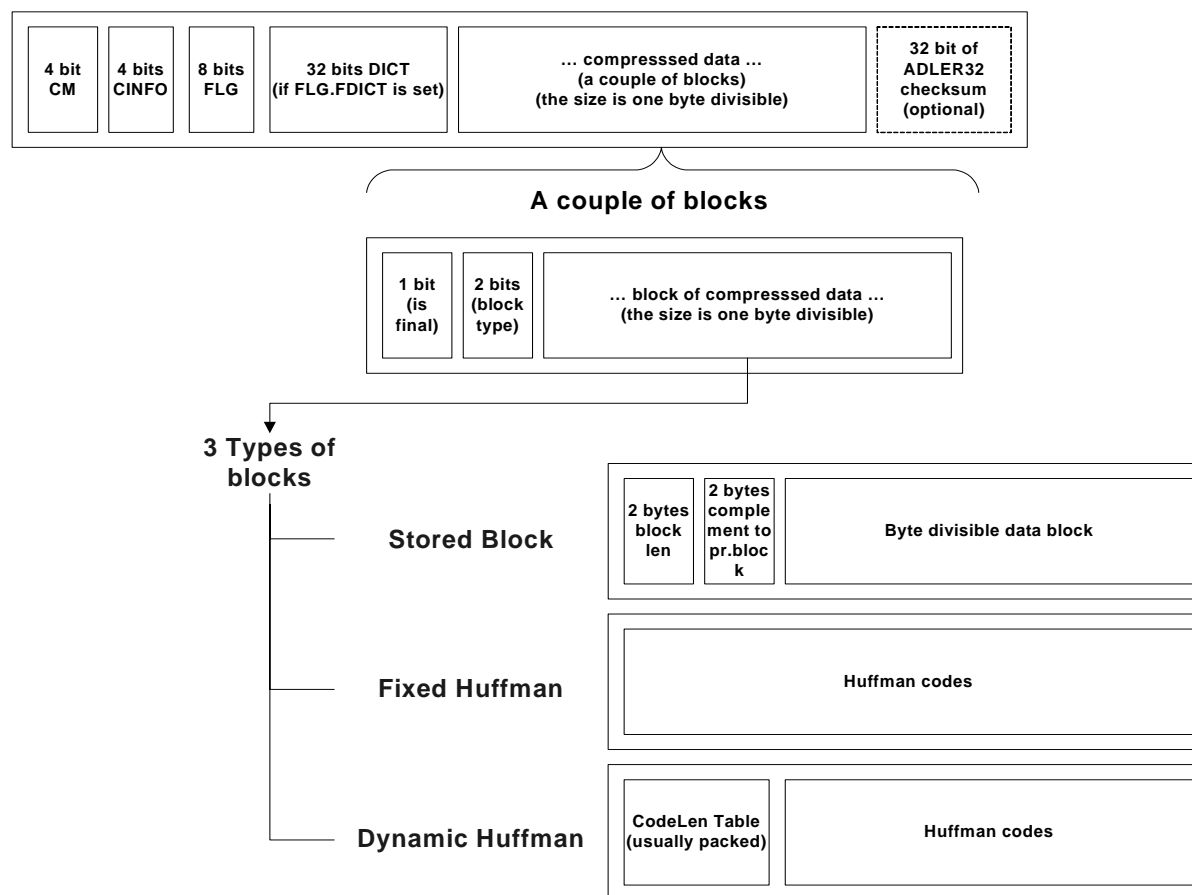
ZLIB Coding Functions

This section describes Intel IPP data compression functions that implement compression methods and data formats defined by the following specifications: [\[RFC1950\]](#), [\[RFC1951\]](#), and [\[RFC1952\]](#). These formats are also known as ZLIB, DEFLATE, and GZIP, respectively.

A basic algorithm for these data compression methods is based on the Lempel-Ziv (LZ77) [\[Ziv77\]](#) dictionary-based compression.

The structure of ZLIB data is schematically shown on the [Figure 13-1](#).

Figure 13-1 ZLIB Data Structure



Figures [Figure 13-2](#) and [Figure 13-3](#) shows the schemes of applying the Intel IPP ZLIB functions for encoding and decoding, respectively. The functions which names are bolded are the mandatory functions for the specified stage.

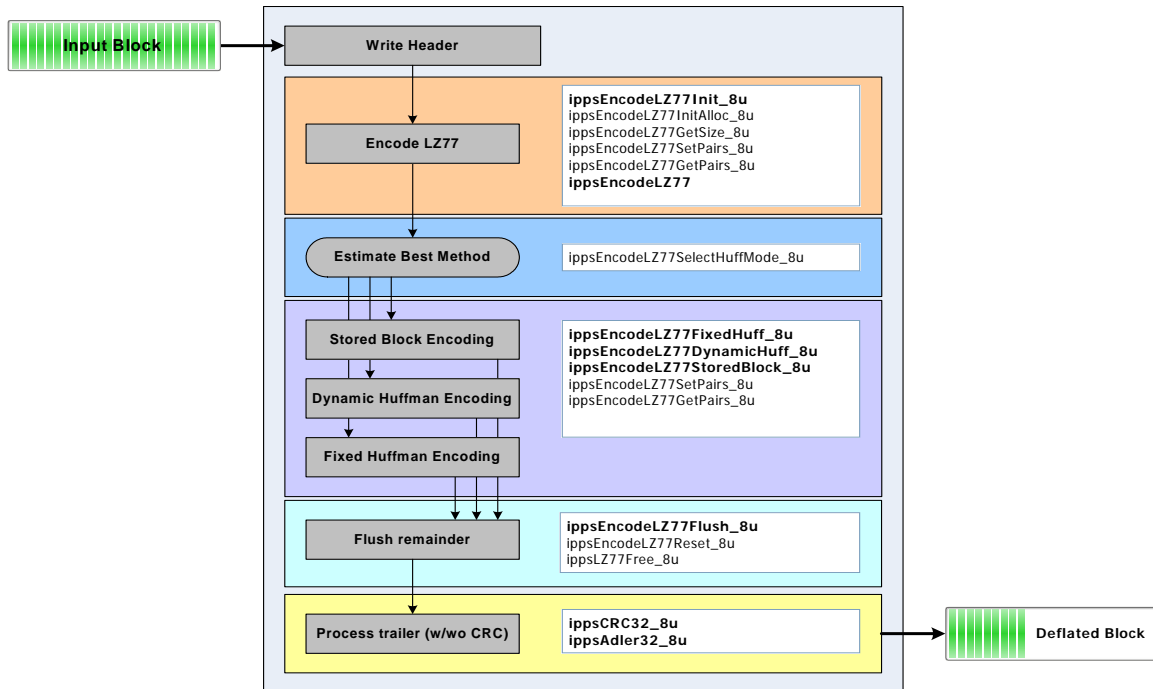
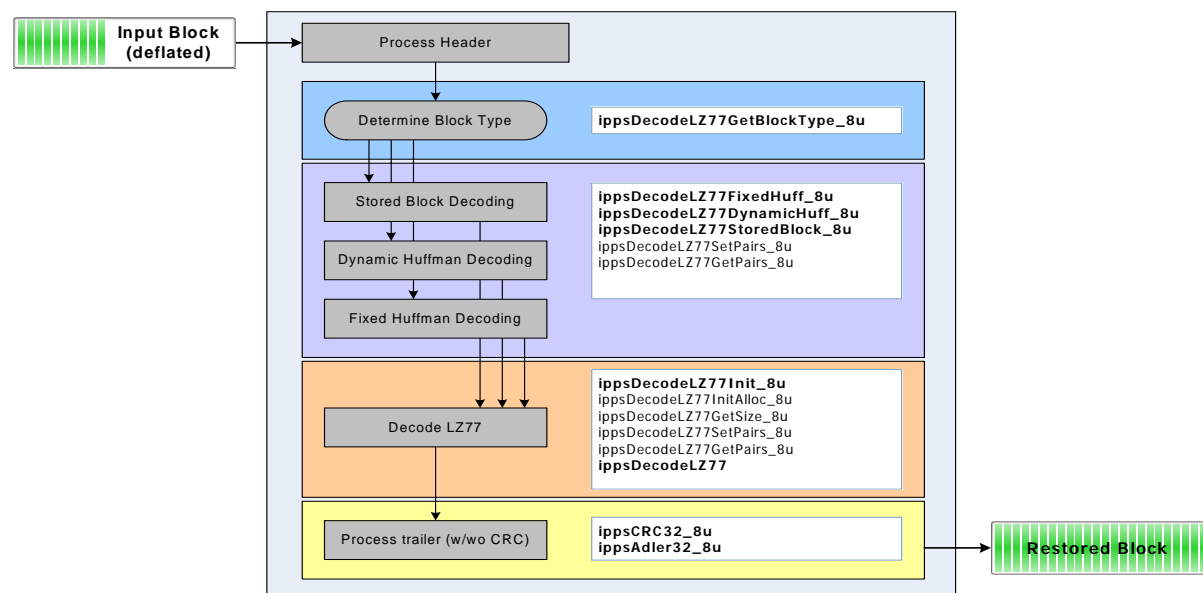
Figure 13-2 Encoding Data Block with Intel IPP ZLIB Functions

Figure 13-3 Decoding Data Block with Intel IPP ZLIB Functions


Special Parameters

The ZLIB coding functions require you to specify several special parameters.

The `comprLevel` parameter specifies the level of compression rate and compression ratio. [Table 13-4](#) lists the possible values of the `comprLevel` parameter and their meanings.

Table 13-4 `comprLevel` Parameter for ZLIB Functions

Value	Descriptions
<code>IppLZ77FastCompr</code>	Fast compression, maximum compression rate, and below average compression ratio
<code>IppLZ77AverageCompr</code>	Average compression rate, average compression ratio
<code>IppLZ77BestCompr</code>	Slow compression, maximum compression ratio

The *checksum* parameter specifies what algorithm is used to compute checksum for input data. [Table 13-5](#) lists the possible values of the *checksum* parameter and their meanings.

Table 13-5 Checksum Parameter for ZLIB Functions

Value	Description
IppLZ77NoChcksm	Checksum is not calculated.
IppLZ77Adler32	Checksum is calculated using Adler32 algorithm.
IppLZ77CRC32	Checksum is calculated using the CRC32 algorithm.

The *flush* parameter specifies the encoding mode for data block encoding. [Table 13-6](#) lists the possible values of the *flush* parameter and their meanings.

Table 13-6 flush Parameter for ZLIB Functions

Value	Descriptions
IppLZ77NoFlush	The end of the block is aligned to a byte boundary.
IppLZ77SyncFlush	The end of the block is aligned to a byte boundary, and 4-byte marker is written to pDst.
IppLZ77FullFlush	The end of the block is aligned to a byte boundary, 4-byte marker is written to pDst, sliding dictionary is zeroed.
IppLZ77FinishFlush	The end of the block is aligned to a byte boundary and the function returns the <code>ippStsStreamEnd</code> status.

The *deflateStatus* parameter specifies the encoding status to ensure the compatibility with the [RFC1951](#) specification. This parameter is used by Intel IPP ZLIB encoding functions. [Table 13-7](#) lists the possible values of the *deflateStatus* parameter and their meanings.

Table 13-7 deflateStatus Parameter for ZLIB Encoding Functions

Value	Descriptions
IppLZ77StatusInit	Specified the deflate implementation of encoding functions, must be used before stream encoding.
IppLZ77StatusLZ77Process	Call the deflate implementation of encoding function.
IppLZ77StatusHuffProcess	Call the deflate implementation of the encoding function with the fixed Huffman codes.
IppLZ77StatusFinal	Specified the last block in the stream.

The *inflateStatus* parameter specifies the decoding status to ensure the compatibility with the [RFC1951](#) specification. This parameter is used by Intel IPP ZLIB decoding functions. [Table 13-8](#) lists the possible values of the *inflateStatus* parameter and their meanings.

Table 13-8 inflateStatus Parameter for ZLIB Decoding Functions

Value	Descriptions
IppLZ77InflateStatusInit	Specified the deflate implementation of encoding functions, must be used before stream encoding.
IppLZ77InflateStatusHuffProcess	Call the deflate implementation of the encoding function with the fixed Huffman codes.
IppLZ77InflateStatusLZ77Process	Call the deflate implementation of encoding function.
IppLZ77InflateStatusFinal	Specified the last block in the stream.

EncodeLZ77Init

Initializes the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77Init_8u (IppLZ77ComprLevel comprLevel, IppLZ77Chcksm
checksum, IppLZ77_8u* pLZ77State);
```

Parameters

<i>comprLevel</i>	Specifies desired compression level (see Table 13-4).
<i>checksum</i>	Specifies an algorithm to compute the checksum for input data (see Table 13-5).
<i>ppLZ77State</i>	Pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77Init` is declared in the `ippdc.h` file. This function initializes the LZ77 encoding state structure in the external buffer. Its size should be previously computed by calling the function [ippsEncodeLZ77GetSize](#). The function implements different algorithms to compute the checksum for input data in accordance with the value of the parameter *checksum* and different compression levels that are specified by the parameter *comprLevel*. The encoding state structure is used by different ZLIB encoder functions.

Alternatively, this state structure can be initialized by the function [ippsEncodeLZ77InitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error if the <code>checksum</code> or <code>comprLevel</code> parameter has an illegal value.

EncodeLZ77GetSize

Computes the size of the LZ77 encoding state structure.

Syntax

```
IppStatus ippEncodeLZ77GetSize_8u (int* pLZ77VLCStateSize);
```

Parameters

`pLZ77StateSize` Pointer to the size of the LZ77 encoding state structure.

Description

The function `ippEncodeLZ77GetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the LZ77 encoding state structure. The function should be called prior to the function [ippEncodeLZ77Init](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pLZ77StateSize</code> is NULL.

EncodeLZ77InitAlloc

Allocates memory and initializes the encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77InitAlloc_8u (IppLZ77comprLevel comprLevel,
    IppLZ77chcksm checksum, IppLZ77_8u** ppLZ77State);
```

Parameters

<i>comprLevel</i>	Specifies desired compression level (see Table 13-4).
<i>checksum</i>	Specifies an algorithm to compute the checksum for input data (see Table 13-5).
<i>ppLZ77State</i>	Double pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77InitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the LZ77 encoding state structure *ppLZ77State*. The function implements different algorithms to compute the checksum for input data in accordance with the value of the parameter *checksum* and different compression levels that are specified by the parameter *comprLevel*. The encoding state structure is used by different ZLIB encoder functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>ppLZ77State</i> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error if the <i>checksum</i> or <i>comprLevel</i> parameter has an illegal value.
<code>ippStsMemAlloc</code>	Indicates an error if memory allocation fails.

LZ77Free

Frees memory allocated for the LZ77 encoding and decoding structures.

Syntax

```
void ippsLZ77Free_8u (IppLZ77State_8u* pLZ77State);
```

Parameters

pLZ77State Pointer to the LZ77 encoding or decoding state structure.

Description

The function `ippsLZ77Free` is declared in the `ippdc.h` file. This function frees memory allocated for the LZ77 encoding or decoding state structures.

EncodeLZ77

Performs LZ77 encoding.

Syntax

```
IppStatus ippsEncodeLZ77_8u (Ipp8u** ppSrc, int* pSrcLen, IppLZ77Pair** ppDst,  
    int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>flush</i>	Specifies the encoding mode for data blocks (see Table 13-6).
<i>pLZ77State</i>	Pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77` is declared in the `ippdc.h` file. This function performs ZLIB encoding [Ziv77] using the sliding window technique in accordance with the RFC1951 specification.

The function uses the LZ77 encoding state structure `pLZ77State` that must be previously initialized by the function [ippsEncodeLZ77InitAlloc](#) or [ippsEncodeLZ77Init](#).

The output data are of `IppLZ77Pair` type that is defined as follows:

```
typedef struct IppLZ77Pairs {  
    Ipp16u length;  
    Ipp16u offset;  
} IppLZ77Pair;
```

where `length` is a match length, and `offset` is a distance between the far-left position of the look-ahead buffer and the first literal of the string match in the sliding window. If the match is not found, the offset field contains 0 and the length field contains a single-literal code.

[Example 13-3](#) and [Example 13-4](#) show how to use the `ippsEncodeLZ77` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>dstLen</code> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

EncodeLZ77SelectHuffMode

Determines the optimal encoding mode.

Syntax

```
IppStatus ippsEncodeLZ77SelectHuffMode_8u(IppLZ77Pair* pSrc, int srcLen,  
    IppLZ77HuffMode* pHuffMode, IppLZ77State_8u* pLZ77State);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Length of the source buffer.
<i>pHuffMode</i>	Pointer to the value of the parameter that specified the optimal encoding mode. Possible values are: IppLZ77UseFixed Encoding with fixed Huffman codes. IppLZ77UseDynamic Encoding with dynamic Huffman codes.
<i>pLZ77State</i>	Pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77SelectHuffMode` is declared in the `ippdc.h` file. This function analyzes the statistics of the source data stream *pSrc* of the [IppLZ77Pair](#) type and determines the optimal encoding mode - with dynamic Huffman codes or with fixed Huffman codes. The function returns the corresponding value of the parameter *pHuffMode*. Source data are produced by the function [ippsEncodeLZ77](#).

This optimal mode is recommended, but you can use your own methods to encode the source data.

[Example 13-4](#) shows how to use the `ippsEncodeLZ77SelectHuffMode` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>pPairs</i> pointer is NULL.

EncodeLZ77FixedHuff

Performs fixed Huffman encoding.

Syntax

```
IppStatus ippsEncodeLZ77FixedHuff_8u (IppLZ77Pair** ppSrc, int* pSrcLen,
    Ipp8u** ppDst, int* pDstLen, IppLZ77flush flush, IppLZ77_8u* pLZ77State);
```

Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
--------------	--------------------------------------

<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>flush</i>	Specifies the encoding mode for data blocks (see Table 13-6).
<i>pLZ77State</i>	Pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77FixedHuff` is declared in the `ippdc.h` file. This function performs encoding with the fixed Huffman codes (see [RFC1951](#) specification) of the input data stream of the `IppLZ77Pair` type that is produced by the function `ippsEncodeLZ77`.

The function `ippsEncodeLZ77FixedHuff` writes the marker of the end of the block to the output stream and aligns the output data in accordance with the specified parameter *flush*.

If the function returns one of the `ippStsNoErr`, `ippStsDstSizeLessExpected`, or `ippStsStreamEnd` statuses, the values of the pointers to the source and destination data buffers and to their lengths are changed and their new values are returned.

The function uses the LZ77 encoding state structure *pLZ77State* that must be previously initialized by the function `ippsEncodeLZ77InitAlloc` or `ippsEncodeLZ77Init`.

[Example 13-4](#) shows how to use the `ippsEncodeLZ77FixedHuff` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.
<code>ippStsStreamEnd</code>	Indicates a warning that the end of the stream occurs. The function returns this status if the parameter <i>flush</i> is set to <code>IppLZ77FinishFlush</code> .

EncodeLZ77DynamicHuff

Performs dynamic Huffman encoding.

Syntax

```
IppStatus ippsEncodeLZ77DynamicHuff_8u(IppLZ77Pair** ppSrc, int* pSrcLen,
    Ipp8u** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u*
    pLZ77State);
```

Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>flush</i>	Specifies the encoding mode for data blocks (see Table 13-6).
<i>pLZ77State</i>	Pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77DynamicHuff` is declared in the `ippdc.h` file. This function performs dynamic Huffman encoding (see [RFC1951](#) specification) of the input data stream of the [IppLZ77Pair](#) type that is produced by the function [ippsEncodeLZ77](#).

The function `ippsEncodeLZ77DynamicHuff` writes the marker of the end of the block to the output stream and aligns the output data in accordance with the specified parameter *flush*.

If the function returns one of the `ippStsNoErr`, `ippStsDstSizeLessExpected`, or `ippStsStreamEnd` statuses, the values of the pointers to the source and destination data buffers and to their lengths are changed and their new values are returned.

The function uses the LZ77 encoding state structure *pLZ77State* that must be previously initialized by the function [ippsEncodeLZ77InitAlloc](#) or [ippsEncodeLZ77Init](#).

[Example 13-4](#) shows how to use the `ippsEncodeLZ77DynamicHuff` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>DstLen</code> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.
<code>ippStsStreamEnd</code>	Indicates a warning that the end of the stream occurs. The function returns this status if the parameter <code>flush</code> is set to <code>IppLZ77FinishFlush</code> .

EncodeLZ77StoredBlock

Stores the data block without compression.

Syntax

```
IppStatus ippEncodeLZ77StoredBlock_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u**  
    ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

Parameters

<code>ppSrc</code>	Double pointer to the source buffer.
<code>pSrcLen</code>	Pointer to the length of the source buffer.
<code>ppDst</code>	Double pointer to the destination buffer.
<code>pDstLen</code>	Pointer to the length of the destination buffer.
<code>flush</code>	Specifies the encoding mode for data blocks (see Table 13-6).
<code>pLZ77State</code>	Pointer to the LZ77 encoding state structure.

Description

The `ippEncodeLZ77StoredBlock` function is declared in the `ippdc.h` file. This function copies source data `ppSrc` data to the destination buffer `ppDst` without Huffman encoding.

The function `ippEncodeLZ77StoredBlock` writes the marker of the end of the block to the output stream and aligns the output data in accordance with the specified parameter `flush`.

The function uses the LZ77 encoding state structure `pLZ77State` that must be previously initialized by the function [ippEncodeLZ77InitAlloc](#) or [ippEncodeLZ77Init](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

EncodeLZ77Flush

Writes the checksum and total length of the input data to the end of the stream.

Syntax

```
IppStatus ippEncodeLZ77Flush_8u (Ipp8u** ppDst, int* pDstLen, IppLZ77State_8u* pLZ77State);
```

Parameters

<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of destination buffer.
<i>pLZ77State</i>	Pointer to the initialized LZ77 encoding state structure.

Description

The `ippEncodeLZ77Flush` function is declared in the `ippdc.h` file. This function writes the computed checksum and total length of the input data stream to the end of the output stream that is encoded by the function [ippEncodeLZ77SelectHuffMode](#). This function is used to ensure compatibility with the [RFC1950](#), [RFC1951](#), and [RFC1952](#) specifications.

[Example 13-4](#) shows how to use the `ippEncodeLZ77Flush` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.

<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.
--	---

EncodeLZ77GetPairs

Retrieves pair data from the LZ77 encoding state structure.

Syntax

```
IppStatus ippEncodeLZ77GetPairs_8u (IppLZ77Pair** ppPairs, int* pPairsInd,
    int* pPairsLen, IppLZ77State_8u* pLZ77State);
```

Parameters

<code>ppPairs</code>	Double pointer to the pair buffer.
<code>pPairsInd</code>	Pointer to the current index in the pair buffer.
<code>pPairsLen</code>	Pointer to the length of the pair buffer.
<code>pLZ77State</code>	Pointer to the initialized LZ77 encoding state structure.

Description

The `ippEncodeLZ77GetPairs` function is declared in the `ippdc.h` file. This function returns the following pointers: `ppPairs` to the pair buffer, `pPairsLen` to the length of the pair buffer, and `pPairsInd` to the current index. These pointers are retrieved from the initialized LZ77 encoding state structure `pLZ77State`.

[Example 13-4](#) shows how the function `ippEncodeLZ77GetPairs` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> or <code>ppPairs</code> pointer is NULL.

EncodeLZ77SetPairs

Sets pair data in the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77SetPairs_8u (IppLZ77Pair* pPairs, int pairsInd, int  
    pairsLen, IppLZ77State_8u* pLZ77State);
```

Parameters

<i>pPairs</i>	Pointer to the pair buffer.
<i>pairsInd</i>	Current index in the pair buffer.
<i>pairsLen</i>	Length of the pair buffer.
<i>pLZ77State</i>	Pointer to the initialized LZ77 encoding state structure.

Description

The `ippsEncodeLZ77SetPairs` function is declared in the `ippdc.h` file. This function sets the pointer to the pair buffer *pPairs*, the length of the pair buffer *pairsLen*, and the current index *pairsInd* in the initialized LZ77 encoding state structure *pLZ77State*.

[Example 13-4](#) shows how the function `ippsEncodeLZ77SetPairs` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>pPairs</i> pointer is NULL.

EncodeLZ77GetStatus

Reads the deflate status value from the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77GetStatus_8u (IppLZ77VLCDeflateStatus* pDeflateStatus,  
    IppLZ77_8u* pLZ77State);
```

Parameters

<i>pDeflateStatus</i>	Pointer to the deflate status parameter (see Table 13-7).
<i>pLZ77State</i>	Pointer to the initialized LZ77 encoding state structure.

Description

The `ippEncodeLZ77GetStatus` function is declared in the `ippdc.h` file. This function reads the current encoding status value *pDeflateStatus* from the initialized LZ77 state structure *pLZ77State*.

[Example 13-4](#) shows how the function `ippEncodeLZ77GetStatus` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>pDeflateStatus</i> pointer is NULL.

EncodeLZ77SetStatus

Sets the deflate status to the desired value in the LZ77 encoding state structure.

Syntax

```
IppStatus ippEncodeLZ77SetStatus_8u (IppLZ77DeflateStatus deflateStatus,
    IppLZ77_8u* pLZ77Stat);
```

Parameters

<i>pLZ77State</i>	Pointer to the initialized LZ77 encoding state structure.
<i>deflateStatus</i>	Specified deflate status value (see Table 13-7).

Description

The `ippEncodeLZ77SetStatus` function is declared in the `ippdc.h` file. This function sets the parameter *deflateStatus* to the specified value in the initialized LZ77 encoding state structure *pLZ77State*.

This parameter should be specified only if the compatibility with the [RFC1950](#), [RFC1951](#), and [RFC1952](#) specifications is required.

[Example 13-4](#) shows how the function `ippsEncodeLZ77SetStatus` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error if the <code>deflateStatus</code> parameter has an illegal value.

EncodeLZ77Reset

Resets the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77Reset_8u (IppLZ77State_8u* pLZ77State);
```

Parameters

<code>pLZ77State</code>	Pointer to the LZ77 encoding state structure.
-------------------------	---

Description

The function `ippsEncodeLZ77Reset` is declared in the `ippdc.h` file. This function resets the LZ77 encoding state structure to the initial state.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> pointer is NULL.

DecodeLZ77Init

Initializes the LZ77 decoding structure.

Syntax

```
IppStatus ippsDecodeLZ77Init_8u(IppLZ77Chcksm checksum, IppLZ77State_8u*
    pLZ77State);
```

Parameters

<i>checksum</i>	Specifies an algorithm to compute the checksum for output data (see Table 13-5).
<i>pLZ77State</i>	Pointer to the initialized LZ77 decoding structure.

Description

The function `ippsDecodeLZ77Init` is declared in the `ippdc.h` file. This function initializes the LZ77 decoding structure in the external buffer whose size should be previously computed by calling the function [ippsDecodeLZ77GetSize](#). The function uses different algorithms to compute the checksum for input data in accordance with the value of the parameter *checksum*. The decoding state structure is used by the function [ippsDecodeLZ77](#).

Alternatively, this structure can be initialized by the function [ippsDecodeLZ77InitAlloc](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>pLZ77State</i> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error if the <i>checksum</i> parameter has an illiegal value.

DecodeLZ77GetSize

Computes the size of the LZ77 decoding structure.

Syntax

```
IppStatus ippsDecodeLZ77GetSize_8u(int* pLZ77StateSize);
```

Parameters

pLZ77StateSize Pointer to the size of the LZ77 decoding structure.

Description

The function `ippsDecodeLZ77GetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the LZ77 decoding state structure. The function should be called prior to the function [ippsDecodeLZ77Init](#).

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error if the pointer *pLZ77StateSize* is NULL.

DecodeLZ77InitAlloc

Allocates memory and initializes the LZ77 decoding structure.

Syntax

```
IppStatus ippsDecodeLZ77InitAlloc_8u(IppLZ77Chcksm checksum, IppLZ77State_8u**  
ppLZ77State);
```

Parameters

checksum Specifies an algorithm to compute the checksum for output data (see [Table 13-5](#)).

ppLZ77State Double pointer to the LZ77 decoding structure.

Description

The function `ippsDecodeLZ77InitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the decoding state structure *ppLZ77State*. The function uses different algorithms to compute the checksum for input data in accordance with the value of the parameter *checksum*. The decoding state structure is used by the function [ippsDecodeLZ77](#).

Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>ppLZ77State</code> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error if the <code>checksum</code> parameter has an illiegal value.
<code>ippStsMemAlloc</code>	Indicates an error if memory allocation fails.

DecodeLZ77

Performs LZ77 decoding.

Syntax

```
IppStatus ippDecodeLZ77_8u(IppLZ77Pair** ppSrc, int* pSrcLen, Ipp8u** ppDst,
    int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

Parameters

<code>ppSrc</code>	Double pointer to the source buffer.
<code>pSrcLen</code>	Pointer to the length of the source buffer.
<code>ppDst</code>	Double pointer to the destination buffer.
<code>pDstLen</code>	Pointer to the length of the destination buffer.
<code>flush</code>	Specifies the encoding mode for data blocks.
<code>pLZ77State</code>	Pointer to the LZ77 decoding structure.

Description

The function `ippDecodeLZ77` is declared in the `ippdc.h` file. This function performs LZ77 decoding of a sequence of pairs of the [IppLZ77Pair](#) type. The sliding window technique implemented in this function ensures compatibility with the [RFC1950](#), [RFC1951](#), and [RFC1952](#) specifications.

The function `ippDecodeLZ77` can be used for decoding data not only in the DEFLATE, ZLIB, GZIP format, but in any other format based on the LZ77 algorithm.

[Example 13-3](#) and [Example 13-5](#) show how to use the `ippDecodeLZ77` function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>ppLZ77State</code> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>DstLen</code> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the destination buffer is full.

DecodeLZ77GetBlockType

Determines the type of encoded data block.

Syntax

```
IppStatus ippDecodeLZ77GetBlockType_8u(Ipp8u** ppSrc, int* pSrcLen,
    IppLZ77HuffMode* pHuffMode, IppLZ77State_8u* pLZ77State);
```

Parameters

<code>ppSrc</code>	Double pointer to the source buffer.
<code>pSrcLen</code>	Pointer to the length of the source buffer.
<code>pHuffMode</code>	Pointer to the value of the parameter that indicates the encoding mode. Possible values are: <div style="display: flex; justify-content: space-between;"> <code>IppLZ77UseFixed</code> Encoding with fixed Huffman codes. </div> <div style="display: flex; justify-content: space-between;"> <code>IppLZ77UseDynamic</code> Encoding with dynamic Huffman codes. </div> <div style="display: flex; justify-content: space-between;"> <code>IppLZ77UseStored</code> Stored block without Huffman coding. </div>
<code>pLZ77State</code>	Pointer to the LZ77 decoding state structure.

Description

The function `ippDecodeLZ77GetBlockType` is declared in the `ippdc.h` file. This function analyzes the source data `ppSrc` and returns the parameter `pHuffMode` which value indicates what Huffman decoding mode must be used for decoding the source bit stream. In accordance with this value the one of the functions [ippDecodeLZ77FixedHuff](#), [ippDecodeLZ77DynamicHuff](#), or [ippDecodeLZ77StoredBlock](#) must be used.



NOTE. If you use only one specified encoding mode, then type of the input data for decoding is known, and this function is not necessary.

[Example 13-5](#) shows how the function `ippsDecodeLZ77GetBlockType` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>ppLZ77State</code> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>SrcLen</code> is less than or equal to 0.
<code>ippStsSrcSizeLessExpected</code>	Indicates a warning that the source buffer is insufficient (internal bit stream and source buffer do not contain enough bits to decode the type of block).

DecodeLZ77FixedHuff

Performs fixed Huffman decoding.

Syntax

```

IppStatus ippsDecodeLZ77FixedHuff_8u(Ipp8u** ppSrc, int* pSrcLen,
    IppLZ77Pair** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u*
    pLZ77State);

```

Parameters

<code>ppSrc</code>	Double pointer to the source buffer.
<code>pSrcLen</code>	Pointer to the length of the source vector.
<code>ppDst</code>	Double pointer to the destination vector.
<code>pDstLen</code>	Pointer to the length of the destination vector.
<code>flush</code>	Specifies the decoding mode for data blocks.
<code>pLZ77State</code>	Pointer to the LZ77 decoding structure.

Description

The function `ippsDecodeLZ77FixedHuff` is declared in the `ippdc.h` file. This function performs decoding of the source data `ppSrc` with the fixed Huffman codes (see [RFC1951](#) specification). The source data must be compatible with the [RFC1950](#), [RFC1951](#), and [RFC1952](#) specifications.

Exit from the `ippsDecodeLZ77FixedHuff` function occurs when:

1. the input stream has ended, but the end-of-block marker is not decoded and the function returns the `ippStsSrcSizeLessExpected` status.
2. the input stream has not ended, but the destination buffer is full and the function returns the `ippStsDstSizeLessExpected` status.
3. the input stream has ended, the end-of-stream marker is decoded and the function returns the `ippStsNoErr` status.

[Example 13-5](#) shows how the function `ippsDecodeLZ77FixedHuff` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsSrcSizeLessExpected</code>	Indicates a warning that the source buffer is less than expected (the end-of-block marker is not encoded).
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the destination buffer is full.
<code>ippStsStreamEnd</code>	Indicates a warning that the stream ends. This warning returns only if the <i>flush</i> parameter is set to <i>IppLZ77FinishFlush</i> .

DecodeLZ77DynamicHuff

Performs dynamic Huffman decoding.

Syntax

```
IppStatus ippsDecodeLZ77DynamicHuff_8u(Ipp8u** ppSrc, int* pSrcLen,
    IppLZ77Pair** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u*
    pLZ77State);
```

Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source vector.

<i>ppDst</i>	Double pointer to the destination vector.
<i>pDstLen</i>	Pointer to the length of the destination vector.
<i>flush</i>	Specifies the decoding mode for data blocks.
<i>pLZ77State</i>	Pointer to the LZ77 decoding structure.

Description

The function `ippsDecodeLZ77DynamicHuff` is declared in the `ippdc.h` file. This function performs decoding of the source data *ppSrc* with the dynamic Huffman codes (see [RFC1951](#) specification).

Exit from the `ippsDecodeLZ77FixedHuff` function occurs when:

1. the input stream has ended, but the end-of-block marker is not decoded and the function returns the `ippStsSrcSizeLessExpected` status.
2. the input stream has not ended, but the destination buffer is full and the function returns the `ippStsDstSizeLessExpected` status.
3. the input stream has ended, the end-of-stream marker is decoded and the function returns the `ippStsNoErr` status.

[Example 13-5](#) shows how the function `ippsDecodeLZ77DynamicHuff` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsSrcSizeLessExpected</code>	Indicates a warning that the source buffer is less than expected (the end-of-block marker is not encoded).
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the destination buffer is full.
<code>ippStsStreamEnd</code>	Indicates a warning that the stream ends. This warning returns only if the <i>flush</i> parameter is set to <i>IppLZ77FinishFlush</i> .

DecodeLZ77StoredBlock

Performs decoding of the block stored without compression.

Syntax

```
IppStatus ippDecodeLZ77StoredBlock_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u**
    ppDst, int* pDstLen, IppLZ77State_8u* pLZ77State);
```

Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source vector.
<i>ppDst</i>	Double pointer to the destination vector.
<i>pDstLen</i>	Pointer to the length of the destination vector.
<i>pLZ77State</i>	Pointer to the LZ77 decoding structure.

Description

The function `ippDecodeLZ77StoredBlock` is declared in the `ippdc.h` file. This function performs decoding of the source data block *ppSrc* that is stored without Huffman coding.

[Example 13-5](#) shows how the function `ippDecodeLZ77StoredBlock` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsWrongBlockType</code>	Indicates a warning that the type of the block is not the “stored without compression type”.
<code>ippStsSrcSizeLessExpected</code>	Indicates a warning that the source buffer is less than expected (the end-of-block marker is not encoded).
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the destination buffer is full.

<code>ippStsStreamEnd</code>	Indicates a warning that the stream ends. This warning returns only if the <i>flush</i> parameter is set to <i>IppLZ77FinishFlush</i> .
------------------------------	---

DecodeLZ77GetPairs

Retrieves pair data from the LZ77 decoding state structure.

Syntax

```
IppStatus ippDecodeLZ77GetPairs_8u(IppLZ77Pair** ppPairs, int* pPairsInd, int* pPairsLen, IppLZ77State_8u* pLZ77State);
```

Parameters

<i>ppPairs</i>	Double pointer to the pair buffer.
<i>pPairsInd</i>	Pointer to the current index in the pair buffer.
<i>pPairsLen</i>	Pointer to the length of the pair buffer.
<i>pLZ77State</i>	Pointer to the initialized LZ77 decoding state structure.

Description

The `ippDecodeLZ77GetPairs` function is declared in the `ippdc.h` file. This function returns the following pointers: *ppPairs* to the pair buffer, *pPairsLen* to the length of the pair buffer, and *pPairsInd* to the current index. These pointers are retrieved from the initialized LZ77 decoding state structure *pLZ77State*.

[Example 13-5](#) shows how the function `ippDecodeLZ77GetPairs` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>ppPairs</i> pointer is NULL.

DecodeLZ77SetPairs

Sets pair data in the LZ77 decoding state structure.

Syntax

```
IppStatus ippDecodeLZ77SetPairs_8u(IppLZ77Pair* pPairs, int pairsInd, int  
    pairsLen, IppLZ77State_8u* pLZ77State);
```

Parameters

<i>pPairs</i>	Pointer to the pair buffer.
<i>pairsInd</i>	Current index in the pair buffer.
<i>pairsLen</i>	Length of the pair buffer.
<i>pLZ77State</i>	Pointer to the initialized LZ77 decoding state structure.

Description

The `ippDecodeLZ77SetPairs` function is declared in the `ippdc.h` file. This function sets the pointer to the pair buffer *pPairs*, the length of the pair buffer *pairsLen*, and the current index *pairsInd* in the initialized LZ77 decoding state structure *pLZ77State*.

[Example 13-5](#) shows how the function `ippDecodeLZ77SetPairs` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>pPairs</i> pointer is NULL.

DecodeLZ77GetStatus

Reads the inflate status value from the LZ77 decoding state structure.

Syntax

```
IppStatus ippDecodeLZ77GetStatus_8u(IppLZ77InflateStatus* pInflateStatus,  
    IppLZ77State_8u* pLZ77State);
```


Parameters

<i>pInflateStatus</i>	Pointer to the inflate status parameter (see Table 13-7).
<i>pLZ77State</i>	Pointer to the LZ77 decoding structure.

Description

The `ippsDecodeLZ77GetStatus` function is declared in the `ippdc.h` file. This function reads the current decoding status value *pInflateStatus* from the initialized LZ77 state structure *pLZ77State*.

[Example 13-5](#) shows how the function `ippsDecodeLZ77GetStatus` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.

DecodeLZ77SetStatus

Sets the inflate status to the desired value in the LZ77 decoding state structure.

Syntax

```
IppStatus ippsDecodeLZ77SetStatus_8u(IppLZ77InflateStatus inflateStatus,
    IppLZ77State_8u* pLZ77State);
```

Parameters

<i>pLZ77State</i>	Pointer to the initialized LZ77 encoding state structure.
<i>inflateStatus</i>	Specified deflate status value (see Table 13-7).

Description

The `ippsDecodeLZ77SetStatus` function is declared in the `ippdc.h` file. This function sets the parameter *inflateStatus* to the specified value in the initialized LZ77 decoding state structure *pLZ77State*.

[Example 13-5](#) shows how the function `ippsDecodeLZ77SetPairs` can be used.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error if the <code>inflateStatus</code> parameter has an illegal value.

DecodeLZ77Reset

Resets the LZ77 decoding state structure.

Syntax

```
IppStatus ippDecodeLZ77Reset_8u(IppLZ77State_8u* pLZ77State);
```

Parameters

`pLZ77State` Pointer to the LZ77 encoding state structure.

Description

The function `ippDecodeLZ77Reset` is declared in the `ippdc.h` file. This function resets the LZ77 decoding state structure to the initial state.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> pointer is NULL.

Adler32

Computes the Adler32 checksum for the source data buffer.

Syntax

```
IppStatus ippAdler32_8u (const Ipp8u* pSrc, int srcLen, Ipp32u* pAdler32);
```

Parameters

<i>pSrc</i>	Pointer to the source data buffer.
<i>srcLen</i>	Number of elements in the source data buffer.
<i>pAdler32</i>	Pointer to the checksum value.

Description

The function `ippsAdler32` is declared in the `ippdc.h` file. This function computes the checksum for *srcLen* elements of the source data buffer *pSrc* and stores it in the *pAdler32*. The checksum is computed using the Adler32 algorithm that is a modified version of the Fletcher algorithm [[Flet82](#)], [[ITU224](#)], [[RFC1950](#)].

You can use this function to compute the accumulated value of the checksum for multiple buffers in the data stream by specifying as an input parameter the checksum value obtained in the preceding function call.

Parameters

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>srcLen</i> is less than or equal to 0.

CRC32

Computes the CRC32 checksum for the source data buffer.

Syntax

```
IppStatus ippsCRC32_8u (const Ipp8u* pSrc, int srcLen, Ipp32u* pCRC32);
```

Parameters

<i>pSrc</i>	Pointer to the source data buffer.
<i>srcLen</i>	Number of elements in the source data buffer.
<i>pCRC32</i>	Pointer to the checksum value.

Description

The function `ippsCRC32` is declared in the `ippdc.h` file. This function computes the checksum for `srcLen` elements of the source data buffer `pSrc` and stores it in the `pCRC32`. The checksum is computed using the CRC32 algorithm [Grif87], [Nel92].

You can use this function to compute the accumulated value of the checksum for multiple buffers in the data stream by specifying as an input parameter the checksum value obtained in the preceding function call.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pSrc</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if the length of the source vector is less than or equal to 0.

Example of Using Intel IPP ZLIB Coding Functions

The code [Example 13-3](#) demonstrates how the model of a compressor can be implemented using the Intel IPP ZLIB functions `ippsEncodeLZ77_8u` and `ippsDecodeLZ77_8u` and an arbitrary bit encoding/decoding methods for the pairs.

Example 13-3 Using the `ippsEncodeLZ77` and `ippsDecodeLZ77` functions

```
#include <stdio.h>
#include <stdlib.h>
#include "ippdc.h"
#define BUFLen 1000
.....
int
main(int argc, char* argv[])
{
    int len;
    int lenr;
    Ipp8u* pSrc;
    int srcLen;
    IppLZ77Pair* pDst;
    int dstLen;
    IppLZ77Pair* pSrcr;
```

Example 13-3 Using the `ippsEncodeLZ77` and `ippsDecodeLZ77` functions

```
int srcLenr;
Ipp8u* pDstr;
int dstLenr;
IppStatus status;
IppLZ77State_8u* pLZ77State;
.....
/* LZ77 encoding */
status = ippsEncodeLZ77InitAlloc_8u(IppLZ77FastCompr, IppLZ77CRC32,
                                     &pLZ77State);
/* Reading the srcLen bytes of input data from the input data file to pSrc */
.....
len = dstLenr;
status = ippsEncodeLZ77_8u(&pSrc, &srcLen, &pDst, &dstLen,
                          IppLZ77FinishFlush, pLZ77State);

/* Here we have the array of Pairs in pDst of (len - dstLen) */
/* In this point the arbitrary bit coding method can be used, */
/* for example LZSS algorithm. */
.....
.....
/* Writing the compressed data to the output file */
ippsLZ77Free_8u(pLZ77State);
/* LZ77 decoding */
status = ippsDecodeLZ77InitAlloc_8u(IppLZ77CRC32, &pLZ77State);

/* Reading the compressed data from the file */
.....
/* On this point the bit decoding method, corresponding to the above */
/* selected bit encoding method (LZSS for example) must be used */
.....
.....
/* After bit decoding the pairs restored placed to the pSrcr of srcLenr */
/* for further performing of LZ77 decoding */
lenr = dstLenr;
status = ippsDecodeLZ77_8u(&pSrcr, &srcLenr, &pDstr, &dstLenr,
                          IppLZ77FinishFlush, pLZ77State);
/* Writing the (lenr - dstLenr) bytes of decompressed data to the output file */
ippsLZ77Free_8u(pLZ77State);

return 0;
} /* main */
```

Implementing the deflate and inflate Functions

The example [Example 13-4](#) demonstrates how the function `deflate` from `zlib.h` library (www.zlib.org) can be implemented using the following Intel IPP functions for ZLIB coding:

- [ippsEncodeLZ77](#),
- [ippsEncodeLZ77FixedHuff](#),
- [ippsEncodeLZ77DynamicHuff](#),
- [ippsEncodeLZ77SelectHuffMode](#),
- [ippsEncodeLZ77Flush](#),
- [ippsEncodeLZ77GetPairs](#),
- [ippsEncodeLZ77SetPairs](#),
- [ippsEncodeLZ77GetStatus](#),
- [ippsEncodeLZ77SetStatus](#).

The example [Example 13-5](#) demonstrates how the function `inflate` from `zlib.h` library (www.zlib.org) can be implemented using the following Intel IPP functions for ZLIB coding:

- [ippsDecodeLZ77](#),
- [ippsDecodeLZ77FixedHuff](#),
- [ippsDecodeLZ77DynamicHuff](#),
- [ippsDecodeLZ77StoredBlock](#),
- [ippsDecodeLZ77GetBlockType](#),
- [ippsDecodeLZ77GetPairs](#),
- [ippsDecodeLZ77SetPairs](#),
- [ippsDecodeLZ77GetStatus](#),
- [ippsDecodeLZ77SetStatus](#).

Note that such functions as `deflateInit`, `inflateInit`, `deflateEnd`, `inflateEnd` from the `zlib.h` library must be implemented using the corresponding Intel IPP functions (`ippsEncodeLZ77InitAlloc`, `ippsDecodeLZ77InitAlloc`, `ippsLZ77Free`) too to use these examples. The full version of the `zlib` library based on the Intel IPP is presented as the Intel IPP Sample downloadable form

<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Example 13-4 Implementing the deflate Function

```
int
deflate(z_stream* strm, int flush)
{
    IppLZ77State_8u* pLZ77State;
    IppLZ77DeflateStatus deflateStatus;
    IppLZ77Flush ippflush;
    IppLZ77Pair* pPairs;
    IppLZ77Pair* cpPairs;
    IppStatus status;
    int pairsInd;
    int pairsLen;
    int psrcLen;
    int pdstLen;
    int srcLen;
    Ipp8u* pSrc;
    int dstLen;
    Ipp8u* pDst;
    IppLZ77HuffMode huffMode;
    int i;

#ifdef GZIP
    unsigned int header;
    unsigned int level_flags;
#endif

    if( flush == Z_NO_FLUSH )
        ippflush = IppLZ77NoFlush;
    else if( flush == Z_SYNC_FLUSH )
        ippflush = IppLZ77SyncFlush;
    else if( flush == Z_FULL_FLUSH )
        ippflush = IppLZ77FullFlush;
    else if( flush == Z_FINISH )
        ippflush = IppLZ77FinishFlush;

    if(strm == Z_NULL || strm->state == Z_NULL || flush > Z_FINISH || flush < 0)
    {
        return Z_STREAM_ERROR;
    }
    pLZ77State = strm->state;
    if(strm->next_out == Z_NULL || (strm->next_in == Z_NULL && strm->avail_in != 0) )
```

Example 13-4 Implementing the deflate Function

```

{
    return Z_STREAM_ERROR;
} for( ; ; )
{
    ippsEncodeLZ77GetStatus_8u(&deflateStatus, pLZ77State);
    if(deflateStatus == IppLZ77StatusInit)
    {
        /* This case should be reached only once, at the beginning of encoding.
*/
        /* Here the standard GZIP/ZLIB header should be written. The type of the
*/
        /* header can be set by defining or undefining the GZIP directive during
*/
        /* the building of the Intel@IPP-based zlib.lib */

        strm->total_out += strm->avail_out;
#ifdef GZIP
        put_byte(strm, 31);
        put_byte(strm, 139);
        put_byte(strm, 8);
        put_byte(strm, 0);
        put_byte(strm, 0);
        put_byte(strm, 0);
        put_byte(strm, 0);
        put_byte(strm, 0);
        put_byte(strm, 4);
        put_byte(strm, 255);
        ippsEncodeLZ77SetStatus_8u(IppLZ77StatusLZ77Process, pLZ77State);
#endif
#ifdef GZIP
        header = (Z_DEFLATED + (7<<4)) << 8;
        level_flags = 2;
        header |= (level_flags << 6);
        header += 31 - (header % 31);
        ippsEncodeLZ77SetStatus_8u(IppLZ77StatusLZ77Process, pLZ77State);
        putShortMSB(strm, header);
        strm->adler = 0;
#endif
        strm->total_out -= strm->avail_out;
    } /* IppLZ77StatusInit */

    ippsEncodeLZ77GetStatus_8u(&deflateStatus, pLZ77State);
    if(deflateStatus == IppLZ77StatusLZ77Process)

```


Example 13-4 Implementing the deflate Function

```
{
    /* In this case deflate processes the LZ77 searches of equal substrings. */

    ippsEncodeLZ77GetPairs_8u(&cpPairs, &pairsInd, &pairsLen, pLZ77State);
    pPairs = cpPairs + pairsInd;
    pdstLen = pairsLen - pairsInd;
    pSrc = strm->next_in;
    srcLen = strm->avail_in;
    strm->total_in += srcLen;
    status = ippsEncodeLZ77_8u(&pSrc, &srcLen, &pPairs, &pdstLen, IPP_FLUSH,
                              pLZ77State);

    strm->total_in -= srcLen;
    strm->next_in = pSrc;
    strm->avail_in = srcLen;
    pairsInd = pairsLen - pdstLen;
    ippsEncodeLZ77SetPairs_8u(cpPairs, pairsInd, pairsLen, pLZ77State);
    if(status < 0 )
        return Z_STREAM_ERROR;
    else if(status == IPP_STS_NO_ERR && IPP_FLUSH == IPP_LZ77_NO_FLUSH)
        return Z_OK;
    else
    {
        deflateStatus = IPP_LZ77_STATUS_HUFF_PROCESS;
        ippsEncodeLZ77SetStatus_8u(deflateStatus, pLZ77State);
    }
} /* IPP_LZ77_STATUS_LZ77_PROCESS */

ippsEncodeLZ77GetStatus_8u(&deflateStatus, pLZ77State);
if(deflateStatus == IPP_LZ77_STATUS_HUFF_PROCESS)
{
    /* In this case deflate estimates the best method for Huffman coding: */
    /* either to use fixed Huffman coding or to use dynamic Huffman coding. */
    /* There are two preprocessor directives, which allows to use only fixed */
    /* Huffman coding without estimation (FORCE_FIXED) or only dynamic */
    /* Huffman coding(FORCE_DYNAMIC). */

    ippsEncodeLZ77GetPairs_8u(&cpPairs, &pairsInd, &pairsLen, pLZ77State);
    psrcLen = pairsLen - pairsInd;
    pPairs = cpPairs + pairsInd;
    pDst = strm->next_out;
    dstLen = strm->avail_out;
    strm->total_out += dstLen;
}
```

Example 13-4 Implementing the deflate Function

```

#ifdef FORCE_FIXED
    Status = ippsEncodeLZ77FixedHuff_8u(&pPairs, &psrcLen, &pDst, &dstLen,
                                         ippflush, pLZ77State);
#endif
#ifndef FORCE_FIXED
#ifdef FORCE_DYNAMIC
    Status = ippsEncodeLZ77DynamicHuff_8u(&pPairs, &psrcLen, &pDst, &dstLen,
                                         ippflush, pLZ77State);
#endif
#endif
#ifndef FORCE_DYNAMIC
    ippsEncodeLZ77SelectHuffMode_8u(pPairs, psrcLen, &huffMode, pLZ77State);
    if(huffMode == IppLZ77UseDynamic)
    {
        status = ippsEncodeLZ77DynamicHuff_8u(&pPairs, &psrcLen, &pDst, &dstLen,
                                             ippflush, pLZ77State);
    }
    else
    {
        status = ippsEncodeLZ77FixedHuff_8u(&pPairs, &psrcLen, &pDst, &dstLen,
                                             ippflush, pLZ77State);
    } /* if else */
#endif
#endif
    strm->total_out -= dstLen;
    strm->next_out = pDst;
    strm->avail_out = dstLen;
    pairsInd = pairsLen - psrcLen;
    if(status < 0)
        return Z_STREAM_ERROR;
    if(status == ippStsDstSizeLessExpected)
    {
        ippsEncodeLZ77SetPairs_8u(cpPairs, pairsInd, pairsLen, pLZ77State);
        return Z_OK;
    }
    else if(status == ippStsNoErr)
    {
        deflateStatus = IppLZ77StatusLZ77Process;
        ippsEncodeLZ77SetStatus_8u(deflateStatus, pLZ77State);
        if(ippflush != IppLZ77FinishFlush) return Z_OK;
    }
    else if(status == ippStsStreamEnd)

```

Example 13-4 Implementing the deflate Function

```

    {
        deflateStatus = IppLZ77StatusFinal;
        ippEncodeLZ77SetStatus_8u(deflateStatus, pLZ77State);
    } /* if else */
} /* IppLZ77StatusHuffProcess */
ippEncodeLZ77GetStatus_8u(&deflateStatus, pLZ77State);
if(deflateStatus == IppLZ77StatusFinal)
{
    /* This case should be reached only once - at the end of encoding.
*/
    /* Here the standard GZIP/ZLIB trailer should be written to the
destination. */

    pDst = strm->next_out;
    dstLen = strm->avail_out;
    strm->total_out += dstLen;
    status = ippEncodeLZ77Flush_8u(&pDst, &dstLen, pLZ77State);
    strm->total_out -= dstLen;
    strm->next_out = pDst;
    strm->avail_out = dstLen;
    if(status < 0 )
        return Z_STREAM_ERROR;
    if(status == ippStsDstSizeLessExpected)
        return Z_OK;
    return Z_STREAM_END;
} /* IppLZ77StatusFinal */

} /* for */
return Z_STREAM_END;
} /* deflate() */

```

Example 13-5 Implementing the inflate Function

```

int
inflate(z_stream* strm, int flush)
{
    int ret;
    unsigned int          in, out;
    IppLZ77State_8u*      pLZ77State;
    IppLZ77InflateStatus  inflateStatus;
    IppStatus              retStatus;
    IppLZ77Flush           ippflush;
    IppLZ77Pair*           pPairConst;
    IppLZ77Pair*           pPair;
    int                    pairsLenConst;
    int                    pairsInd;
    Ipp8u*                 pSrc;
    int                    srcLen;
    Ipp8u*                 pDst;
    int                    dstLen;
    IppLZ77HuffMode        huffMode;
    int                    i;
    int                    copy;
    unsigned int           len;

    in = strm->avail_in;
    out = strm->avail_out;
    pLZ77State = (IppLZ77State_8u*) strm->state;
    ippflush = IppLZ77NoFlush;

    for(;;)
    {
        ippDecodeLZ77GetStatus_8u(&inflateStatus, pLZ77State);
        ippDecodeLZ77GetPairs_8u(&pPairConst, &pairsInd, &pairsLenConst,
pLZ77State);
        if(inflateStatus == IppLZ77InflateStatusInit)
        {
            /* This case should be reached only once at the beginning of decoding. */
            /* Processing standard GZIP or ZLIB header.                               */
            .....
            inflateStatus = IppLZ77InflateStatusHuffProcess;
            ippDecodeLZ77SetStatus_8u(inflateStatus, pLZ77State);

```

Example 13-5 Implementing the inflate Function

```
    } /* IppLZ77InflateStatusInit */
    else if(inflateStatus == IppLZ77InflateStatusHuffProcess)
    {
        /* This case performs Huffman decoding of the deflate format (rfc1951.txt).*/
        /* Before decoding the ippsDecodeLZ77GetBlockType_8u is called to      */
        /* determine the type of Huffman decoding should be used: fixed or      */
        /* dynamic Huffman decoding, or copying uncoded stored block            */
        /* from input to the output.                                             */

        pSrc = strm->next_in;
        srcLen = strm->avail_in;
        retStatus = ippsDecodeLZ77GetBlockType_8u(&pSrc, &srcLen, &huffMode,
                                                  pLZ77State);

        strm->avail_in = srcLen;
        strm->next_in = pSrc;
        if(retStatus == ippStsNoErr)
        {
            /* ippsDecodeLZ77GetBlockType_8u returned ippStsNoErr.  */
            /* It means the block type was determined successfully. */

            if(huffMode == IppLZ77UseFixed)
            {
                /* The type of block determined by ippsDecodeLZ77GetBlockType_8u is */
                /* IppLZ77UseFixed.                                                  */

                pPair = pPairConst + pairsInd;
                dstLen = pairsLenConst - pairsInd;
                pSrc = strm->next_in;
                srcLen = strm->avail_in;

                retStatus = ippsDecodeLZ77FixedHuff_8u(&pSrc, &srcLen, &pPair,
                                                       &dstLen, ippflush, pLZ77State);

                strm->next_in = pSrc;
                strm->avail_in = srcLen;
                pairsInd = pPair - pPairConst;
                if(retStatus == ippStsNoErr)
                {
                    ippsDecodeLZ77SetPairs_8u(pPairConst, pairsInd, pairsLenConst,
                                              pLZ77State);

                    if(flush == Z_BLOCK)
                    {
                        strm->blockEnd = 1;
                        inflateStatus = IppLZ77InflateStatusLZ77Process;
                    }
                }
            }
        }
    }
}
```

Example 13-5 Implementing the inflate Function

```

        ippsDecodeLZ77SetPairs_8u(pPairConst, 0, pairsInd, pLZ77State);
    } /* if */
}
elseif(retStatus==ippStsDstSizeLessExpected||retStatus==ippStsStreamEnd)
{
    inflateStatus = IppLZ77InflateStatusLZ77Process;
    ippsDecodeLZ77SetPairs_8u(pPairConst, 0, pairsInd, pLZ77State);
    if(retStatus == ippStsStreamEnd)
        strm->headerMode = infhead;
}
else if(retStatus == ippStsSrcSizeLessExpected)
{
    ippsDecodeLZ77SetPairs_8u(pPairConst, pairsInd, pairsLenConst,
                             pLZ77State);

    ret = Z_OK;
    goto inf_leave;
} /* if else */
    ippsDecodeLZ77SetStatus_8u(inflateStatus, pLZ77State);
}
else if(huffMode == IppLZ77UseDynamic)
{
    /* The type of block determined by ippsDecodeLZ77GetBlockType_8u is */
    /* IppLZ77UseDynamic. */

    pPair = pPairConst + pairsInd;
    dstLen = pairsLenConst - pairsInd;
    pSrc = strm->next_in;
    srcLen = strm->avail_in;
    retStatus = ippsDecodeLZ77DynamicHuff_8u(&pSrc, &srcLen, &pPair, &dstLen,
                                             ippflush, pLZ77State);

    strm->next_in = pSrc;
    strm->avail_in = srcLen;
    pairsInd = pPair - pPairConst;

    if(retStatus == ippStsNoErr)
    {
        ippsDecodeLZ77SetPairs_8u(pPairConst, pairsInd, pairsLenConst,
                                   pLZ77State);

        if(flush == Z_BLOCK)
        {
            strm->blockEnd = 1;
            inflateStatus = IppLZ77InflateStatusLZ77Process;

```

Example 13-5 Implementing the inflate Function

```
        ippDecodeLZ77SetPairs_8u(pPairConst, 0, pairsInd, pLZ77State);
    } /* if */
}
elseif(retStatus==ippStsDstSizeLessExpected||retStatus==ippStsStreamEnd)
{
    inflateStatus = IppLZ77InflateStatusLZ77Process;
    ippDecodeLZ77SetPairs_8u(pPairConst, 0, pairsInd, pLZ77State);
    if(retStatus == ippStsStreamEnd)
        strm->headerMode = infhead;
}
else if(retStatus == ippStsSrcSizeLessExpected)
{
    ippDecodeLZ77SetPairs_8u(pPairConst, pairsInd, pairsLenConst,
                           pLZ77State);

    ret = Z_OK;
    goto inf_leave;
} /* if else */
    ippDecodeLZ77SetStatus_8u(inflateStatus, pLZ77State);
}
else if(huffMode == IppLZ77UseStored)
{
    /* The type of block determined by ippDecodeLZ77GetBlockType_8u is */
    /* IppLZ77UseStored. */

    pSrc = strm->next_in;
    srcLen = strm->avail_in;
    pDst = strm->next_out;
    dstLen = strm->avail_out;
    retStatus = ippDecodeLZ77StoredBlock_8u(&pSrc, &srcLen, &pDst,
                                           &dstLen, pLZ77State);

    strm->next_in = pSrc;
    strm->avail_in = srcLen;
    strm->next_out = pDst;
    strm->avail_out = dstLen;
    if(retStatus == ippStsNoErr)
    {
        inflateStatus = IppLZ77InflateStatusHuffProcess;
    }
    else if(retStatus == ippStsStreamEnd)
    {
        inflateStatus = IppLZ77InflateStatusFinal;
        strm->headerMode = infhead;
    }
}
```

Example 13-5 Implementing the inflate Function

```

        else if(retStatus == ippStsDstSizeLessExpected || retStatus ==
ippStsSrcSizeLessExpected)
        {
            ret = Z_OK;
            goto inf_leave;
        }
        else
        {
            ret = Z_DATA_ERROR;
            goto inf_leave;
        } /* if else */
        ippDecodeLZ77SetStatus_8u(inflateStatus, pLZ77State);
    } /* if else */
}
else if(retStatus == ippStsSrcSizeLessExpected)
{
    /* ippDecodeLZ77GetBlockType_8u returned ippStsSrcSizeLessExpected. */
    /* It means that there is not enough bits in bitstream to decode */
    /* the type of block. */

    ret = Z_OK;
    goto inf_leave;
}
else
{
    ret = Z_DATA_ERROR;
    goto inf_leave;
} /* if else */
} /* IppLZ77InflateStatusHuffProcess */
else if(inflateStatus == IppLZ77InflateStatusLZ77Process)
{
    /* This case performs LZ77 decoding of the data in the buffer of Pairs. */

    pPair = pPairConst + pairsInd;
    srcLen = pairsLenConst - pairsInd;
    pDst = strm->next_out;
    dstLen = strm->avail_out;
    retStatus = ippDecodeLZ77_8u(&pPair, &srcLen, &pDst, &dstLen, ippflush,
                                pLZ77State);

    strm->next_out = pDst;
    strm->avail_out = dstLen;
    pairsInd = pPair - pPairConst;
    if(retStatus == ippStsNoErr)

```

Example 13-5 Implementing the inflate Function

```
{
    ippDecodeLZ77SetPairs_8u(pPairConst, 0, pairsInd, pLZ77State);
    inflateStatus = IppLZ77InflateStatusHuffProcess;
    if((flush == Z_BLOCK) && (strm->blockEnd == 1))
    {
        ippDecodeLZ77SetStatus_8u(inflateStatus, pLZ77State);
        ret = Z_OK;
        strm->blockEnd = 0;
        goto inf_leave;
    } /* if */
}
else if(retStatus == ippStsDstSizeLessExpected)
{
    ippDecodeLZ77SetPairs_8u(pPairConst, pairsInd, pairsLenConst, pLZ77State);
    ret = Z_OK;
    goto inf_leave;
}
else if(retStatus == ippStsStreamEnd)
{
    ippDecodeLZ77SetPairs_8u(pPairConst, 0, pairsInd, pLZ77State);
    inflateStatus = IppLZ77InflateStatusFinal;
}
else
{
    ret = Z_DATA_ERROR;
    goto inf_leave;
} /* if else */
    ippDecodeLZ77SetStatus_8u(inflateStatus, pLZ77State);
} /* IppLZ77InflateStatusLZ77Process */

else if(inflateStatus == IppLZ77InflateStatusFinal)
{
    /* This case should be reached only once: when the end of deflate stream */
    /* is decoded. This case processes the standart GZIP/ZLIB trailer.      */
    .....

    strm->next_in -= ((int)strm->avail_in);
    strm->avail_in = 0;
    ret = Z_STREAM_END;
    goto inf_leave;
} /* IppLZ77InflateStatusFinal */
```

Example 13-5 Implementing the `inflate` Function

```
        else
        {
            ret = Z_DATA_ERROR;
            goto inf_leave;
        } /* if else */
    } /* for */

inf_leave:

    in -= strm->avail_in;
    out -= strm->avail_out;
    strm->total_in += in;
    strm->total_out += out;

    return ret;
} /* inflate() */
```

BWT-Based Compression Functions

This section describes the Intel IPP functions that support composed algorithms based on the Burrows-Wheeler transform (BWT). The full list of these functions is given in the [Table 13-9](#).

Table 13-9 Intel IPP BWT-Based Compression Functions

Function Short Name	Description
BWT Functions	
BUTFwdGetSize	Computes the size of the external buffer for the forward BWT transform.
BUTFwd	Performs the forward BWT transform.
BWTInvGetSize	Computes the size of the external buffer for the inverse BWT transform.
BWTInv	Performs the inverse BWT transform.
BWTGetSize_SmalBlock	Computes the size of the external buffer for the BWT transforms for small data block.
BUTFwd_SmallBlock	Performs the forward BWT transform for small data block.
BWTInv_SmallBlock	Performs the inverse BWT transform for small data block.
GIT Functions	
EncodeGITInitAlloc	Allocates memory and initializes the GIT encoding state structure.
GITFree	Frees memory allocated by the function <code>EncodeGITInitAlloc</code> or <code>DecodeGITInitAlloc</code> .
EncodeGITInit	Initializes the GIT encoding state structure.
EncodeGITGetSize	Computes the size of the GIT encoding state structure.
EncodeGIT	Performs GIT encoding.
DecodeGITInitAlloc	Allocates memory and initializes the GIT decoding state structure.
DecodeGITInit	Initializes the GIT decoding state structure.
DecodeGITGetSize	Computes the size of the GIT decoding state structure.
DecodeGIT	Performs GIT decoding.
MTF Functions	
MTFInitAlloc	Allocates memory and initializes the MTF structure.
MTFFree	Frees memory allocated for the MTF structure.
MTFInit	Initializes the MTF structure.
MTFGetSize	Computes the size of the MTF structure.
MTFFwd	Performs the forward MTF transform.
MTFInv	Performs the inverse MTF transform.

Table 13-9 Intel IPP BWT-Based Compression Functions (continued)

Function Short Name	Description
RLE Functions	
EncodeRLE	Performs RLE encoding.
DecodeRLE	Performs RLE decoding.

Burrows-Wheeler Transform

This section describes data compression functions that perform Burrows-Wheeler Transform (BWT). This method does not compress data, but it simplifies the structure of input data and makes more effective further compression. One of the distinctive feature of this method is operation on the block of data (as a rule of size 512kB - 2 mB). The main idea of this method is block sorting which groups symbols with a similar context. Let us consider how BWT works on the input data block ‘abracadabra’. The first step is to create a matrix containing all its possible cyclic permutations. The first row is input string, the second is created by shifting it to the left by one symbol and so on:

abracadabra
bracadabraa
racadabraab
acadabraabr
cadabraabra
adabraabrac
dabraabraca
abraabracad
braabracada
raabracadab
aabracadabr

Then all rows are sorted in accordance with the lexicographic order:

0 aabracadabr
1 abraabracad
2 abracadabra
3 acadabraabr
4 adabraabrac
5 braabracada
6 bracadabraa
7 cadabraabra
8 dabraabraca
9 raabracadab
10 racadabraab

The last step is to write out the last column and the index of the input string: `rdarcaaaabb`, 2 - this is a result of the forward BWT transform.

Inverse BTW is performed as follows:

elements of the input string are numbered in ascending order

```

0  r
1  d
2  a
3  r
4  c
5  a
6  a
7  a
8  a
9  b
10 b

```

and sorted in accordance with the lexicographic order:

```

2  a
5  a
6  a
7  a
8  a
9  b
10 b
4  c
1  d
0  r
3  r

```

This index array is a vector of the inverse transform (`Inv`), the further reconstruction of the string is performed in the following manner:

```

src[] = "rdarcaaaabb";
Inv[] = {2,5,6,7,8,9,10,4,1,0,3};
index = 2; // index of the initial string is known from the forward BWT
for( i = 0; i < len; i++ ) {
    index = Inv[index];
    dst[i] = src[index];
}

```

BWTFwdGetSize

Computes the size of the external buffer for the forward BWT transform.

Syntax

```
IppStatus ippsBWTFwdGetSize_8u(int wndSize, int* pBWTFwdBuffSize);
```

Parameters

<i>wndSize</i>	Window size for BWT transform.
<i>pBWTFwdStateSize</i>	Pointer to the computed size of the additional buffer.

Description

The function `ippsBWTFwdGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the external buffer that is required by the function [ippsBWTFwd](#) for the forward BWT transform.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pBWTFwdBuffSize</i> pointer is NULL.

BWTFwd

Performs the forward BWT transform.

Syntax

```
IppStatus ippsBWTFwd_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,  
    int* pIndex, Ipp8u* pBWTFwdBuff);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.

<i>len</i>	Number of elements in the source and destination vectors.
<i>pIndex</i>	Pointer to the index of first position for the forward BWT transform.
<i>pBWTfwdBuff</i>	Pointer to the additional buffer.

Description

The function `ippsBWTfwd` is declared in the `ippdc.h` file. This function performs the forward BWT transform of *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTfwdBuff*. The size of this buffer should be computed by calling the function [ippsBWTfwdGetSize](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.

BWTInvGetSize

Computes the size of the external buffer for the inverse BWT transform.

Syntax

```
IppStatus ippsBWTInvGetSize_8u(int wndSize, int* pBWTInvBuffSize);
```

Parameters

<i>wndSize</i>	Window size for BWT transform.
<i>pBWTInvBuffSize</i>	Pointer to the computed size of the additional buffer.

Description

The function `ippsBWTInvGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the external buffer that is required by the function [ippsBWTInv](#) for the inverse BWT transform.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <code>pBWTInvBuffSize</code> pointer is NULL.

BWTInv

Performs the inverse BWT transform.

Syntax

```
IppStatus ippSBWTInv_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int index,  
                        Ipp8u* pBWTInvBuff);
```

Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>len</code>	Number of elements in the source and destination vectors.
<code>index</code>	Index of first position for the inverse BWT transform.
<code>pBWTInvBuff</code>	Pointer to the additional buffer.

Description

The function `ippSBWTInv` is declared in the `ippdc.h` file. This function performs the inverse BWT transform of `len` elements starting from `pIndex` element of the source vector `pSrc` and stores result in the vector `pDst`. The function uses the external buffer `pBWTInvBuff`. The size of this buffer should be computed by calling the function [ippSBWTInvGetSize](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>len</code> is less than or equal to 0.

BWTGetSize_SmalBlock

Computes the size of the external buffer for the BWT transforms for small data block.

Syntax

```
IppStatus ippsBWTGetSize_SmallBlock_8u(int wndSize, int* pBuffSize);
```

Parameters

<i>wndSize</i>	Window size for BWT transform.
<i>pBuffSize</i>	Pointer to the computed size of the additional buffer.

Description

The function `ippsBWTGetSize_SmallBlock` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the external buffer that is required for the forward and inverse BWT transforms of the small data block by the functions [ippsBUTFwd_SmallBlock](#) and [ippsBWTInv_SmallBlock](#) respectively. The size of the data block must be in the range (0, 32768].

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pBuffSize</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>wndSize</i> is less than or equal to 0, or is greater than 32768.

BUTFwd_SmalBlock

Performs the forward BWT transform for small data block.

Syntax

```
IppStatus ippsBUTFwd_SmallBlock_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,
    int* pIndex, Ipp8u* pBWTBuff);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the source and destination vectors.
<i>pIndex</i>	Pointer to the index of first position for the forward BWT transform.
<i>pBWTBuff</i>	Pointer to the additional buffer.

Description

The function `ippsBWTFwd_SmallBlock` is declared in the `ippdc.h` file. This function performs the forward BWT transform of small block of data, that the number of elements must be in the range (0, 32768]. The function processes *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTBuff*. The size of this buffer must be computed by calling the function [ippsBWTGetSize_SmalBlock](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0, or is greater than 32768.

BWTInv_SmallBlock

Performs the inverse BWT transform for small data block.

Syntax

```
IppStatus ippsBWTInv_SmallBlock_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int index, Ipp8u* pBWTBuff);
```

Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.

<i>len</i>	Number of elements in the source and destination vectors.
<i>index</i>	Index of first position for the inverse BWT transform.
<i>pBWTBuff</i>	Pointer to the additional buffer.

Description

The function `ippsBWTInv_SmallBlock` is declared in the `ippdc.h` file. This function performs the inverse BWT transform of small block of data, that the number of elements must be in the range (0, 32768]. The function processes *len* elements starting from *index* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTBuff*. The size of this buffer must be computed by calling the function [ippsBWTGetSize_SmalBlock](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0, or is greater than 32768; or if <i>index</i> is greater than or equal to .

Generalized Interval Transformation Coding

This section describes Intel IPP data compression functions that perform Generalized Interval Transformation (GIT) coding. The coding procedure is of a statistical entropy nature. The GIT algorithm is based on Interval Transformations also known as Distance coding or Inversion frequencies [[Arn97](#)] and Rice-Golomb coding [[Gal75](#)]. This algorithm compresses blocks of data at ratios that are close to Huffman encoding compression ratios. However, with low entropy source data, the GIT algorithm typically provides coding at higher compression ratios as compared with the Huffman algorithm.

Special Arguments

The GIT coding functions give you possibility to adjust the compression algorithm specifying the *strategyHint* argument. The *strategyHint* argument suggests using different type of reordering the symbols in the alphabet in dependence on the input data. In some cases it can improve encoding. [Table 13-10](#) lists the possible values of the *strategyHint* argument and their meanings.

Table 13-10 strategyHint Argument for GIT Encoding Functions

Value	Descriptions
ippGITNoStrategy	No strategy.
ippGITLeftReorder	Alphabet is reordered in such a way that more frequent symbols have less numbers (moved to the left).
ippGITRightReorder	Alphabet is reordered in such a way that more frequent symbols have greater numbers (move to right)
ippGITFixedOrder	Fixed alphabet order, no reordering

EncodeGITInitAlloc

Allocates memory and initializes the GIT encoding state structure.

Syntax

```
IppStatus ippEncodeGITInitAlloc_8u (int maxSrcLen, int maxDstLen,
    IppGITState_8u** ppGITState);
```

Parameters

- maxSrcLen* Maximum length of the source buffer.
- maxDstLen* Maximum length of the destination buffer.
- ppGITState* Double pointer to the GIT encoding state structure.

Description

The function `ippEncodeGITInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the GIT encoding state structure *ppGITState*. This structure is used by the function [ippEncodeGIT](#).

The size of the GIT encoding state structure depends on the values of the *maxSrcLen* and *maxDstLen* parameters that set the upper bound on the sizes of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

The *maxSrcLen* parameter sets the size of a block for input data processing. Due to the adaptive coding used by the GIT algorithm, the compression ratio is very likely to improve as the block size increases.

Return Values

<code>IppStsNoErr</code>	Indicates no errors.
<code>IppStsSizeErr</code>	Indicates an error if <i>maxSrcLen</i> or <i>maxDstLen</i> is less than or equal to 0.

GITFree

Frees memory allocated for the GIT coding structures.

Syntax

```
void ippsGITFree_8u (IppGITState_8u* pGITState);
```

Parameters

<i>pGITState</i>	Pointer to the GIT encoding or decoding state structure.
------------------	--

Description

The function `ippsGITFree` is declared in the `ippdc.h` file. This function frees memory allocated for the GIT encoding or decoding state structure by the functions [ippsEncodeGITInitAlloc](#) or [ippsDecodeGITInitAlloc](#), respectively.

EncodeGITInit

Initializes the GIT encoding state structure.

Syntax

```
IppStatus ippsEncodeGITInit_8u (int maxSrcLen, int maxDstLen,
    IppGITState_8u* pGITState);
```

Parameters

<i>maxSrcLen</i>	Maximum length of the source buffer.
<i>maxDstLen</i>	Maximum length of the destination buffer.
<i>pGITState</i>	Pointer to the GIT encoding state structure.

Description

The function `ippsEncodeGITInit` is declared in the `ippdc.h` file. This function initializes the GIT encoding state structure in the external buffer whose size should be computed previously by the calling the function [ippsEncodeGITGetSize](#). Alternatively this structure can be initialized by the function [ippsEncodeGITInitAlloc](#).

The size of the GIT encoding state structure depends on the values of the *maxSrcLen* and *maxDstLen* parameters, which set the upper bound on the sizes of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

The *maxSrcLen* parameter sets the size of a block for input data processing. Due to adaptive coding used by the GIT algorithm, the compression ratio is very likely to improve as the block size increases.



NOTE. The *maxSrcLen* and *maxDstLen* parameters of the `ippsEncodeGITInit` and [ippsEncodeGITGetSize](#) functions must have equal respective values to avoid errors in the [ippsEncodeGIT](#) function.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pGITState</code> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>maxSrcLen</code> or <code>maxDstLen</code> is less than or equal to 0.

EncodeGITGetSize

Computes the size of the GIT encoding state structure.

Syntax

```
IppStatus ippEncodeGITGetSize_8u (int maxSrcLen, int maxDstLen, int*
    pGITStateSize);
```

Parameters

<code>maxSrcLen</code>	Maximum length of the source buffer.
<code>maxDstLen</code>	Maximum length of the destination buffer.
<code>pGITStateSize</code>	Pointer to the size of the GIT encoding state structure.

Description

The function `ippEncodeGITGetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the GIT encoding state structure. Its size depends on the `maxSrcLen` and `maxDstLen` parameters, which are the maximum lengths of the input and output data buffers, respectively.

The function should be called prior to the function [ippEncodeGITInit](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pGITStateSize</code> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>maxSrcLen</code> or <code>maxDstLen</code> is less than or equal to 0.

EncodeGIT

Performs GIT encoding.

Syntax

```
IppStatus ippsEncodeGIT_8u (const Ipp8u* pSrc, int srcLen, Ipp8u* pDst,
    int* pDstLen, IppGITStrategyHint strategyHint, IppGITState_8u* pGITState);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Length of the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>strategyHint</i>	Suggests using of the code implemented specific strategy for lexicographical reordering (see Table 13-10); default value is <code>ippGITNoStrategyHint</code> .
<i>pGITState</i>	Pointer to the GIT encoding state structure.

Description

The function `ippsEncodeGIT` is declared in the `ippdc.h` file. This function performs GIT encoding. It processes the *srcLen* bytes of the input data *pSrc* and writes the results to the *pDst* buffer. The function uses the GIT encoding state structure *pGITState* that contains pointers to different data structures required for operation. The structure must be initialized by the functions [ippsEncodeGITInitAlloc](#) or [ippsEncodeGITInit](#) beforehand.

It is not recommended to use source data buffer less than 8Kb.



WARNING. Value of *srcLen* must not exceed that of *maxSrcLen* specified in the function [ippsEncodeGITInitAlloc](#) or [ippsEncodeGITInit](#).

Nevertheless, the larger files can be encoded by subsequently calling the `ippsEncodeGIT` function several times (see [Example 13-6](#) below).

Encoding quality depends on the input data and in some cases can be improved by specifying the *strategyHint* parameter (see [Table 13-10](#)).

The default value for the *strategyHint* parameter is `ippGITNoStrategyHint`.

The code [Example 13-6](#) shows how to use the `ippEncodeGIT_8u` and supporting functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>srcLen</i> is greater than the value of the <i>maxSrcLen</i> parameter passed to ippEncodeGITGetSize or ippEncodeGITInitAlloc .

Example 13-6 Using the Function `EncodeGIT` and Supporting Functions

```

#define BLOCKSIZE 65536
.....
FILE*          in, out;
Ipp8u*         data, code;
IppGITState_8u* pGITState;
int            GITStateSize, size, codeLen;
IppGITStrategyHint strategyHint;
.....

/*****/
/* Opening the file containing input data and the file      */
/* for the compressed data:                                */
/*****/
in = fopen("datafile.txt", "rb");
out = fopen("codefile", "wb");

/*****/
/* Memory allocation for input data and output code buffers, */
/* and for the internal encoding state structure:            */
/*****/
data = (Ipp8u*) malloc(BLOCKSIZE * sizeof(Ipp8u));
code = (Ipp8u*) malloc((BLOCKSIZE << 1) * sizeof(Ipp8u));
ippsEncodeGITGetSize_8u(BLOCKSIZE, (BLOCKSIZE << 1), &GITStateSize);
pGITState = (IppGITState_8u*) malloc(GITStateSize * sizeof(Ipp8u));

/*****/
/* Initializing the memory, allocated for internal encoding */
/* state structure and setting the encoding strategy:        */
/*****/
ippsEncodeGITInit_8u(BLOCKSIZE, (BLOCKSIZE << 1), pGITState);
strategyHint = ippGITLeftReorder;

```

Example 13-6 Using the Function `EncodeGIT` and Supporting Functions

```

/*****
/* The main loop. In every iteration program reads the size
/* bytes from the input data file to data array and
/* compresses by calling the ippsEncodeGIT_8u function.
/* ippsEncodeGIT_8u places the compressed data to the code
/* array and returns the length of compressed data in bytes
/* through the &codeLen argument. After that program writes
/* the value of codeLen and codeLen bytes of compressed data
/* to the output data file.
*****/
for( ; ; )
{
    size = fread(&data[0], sizeof(Ipp8u), BLOCKSIZE, in);
    if(size <= 0)
        break;
    if(ippsEncodeGIT_8u(data, size, code, &codeLen, strategyHint,
pGITState) != ippsStsNoErr)
        return (-1);
    fwrite(&codeLen, sizeof(int), 1, out);
    fwrite(code, sizeof(Ipp8u), codeLen, out);
} /* for */

/*****
/* Free the memory allocated for input data and output
/* code buffers and for the internal encoding state
/* structure:
*****/
free(pGITState);
free(data);
free(code);
fclose(in);
fclose(out);
.....

```

DecodeGITInitAlloc

Allocates memory and initializes the GIT decoding state structure.

Syntax

```
IppStatus ippsDecodeGITInitAlloc_8u (int maxSrcLen, int maxDstLen,  
    IppGITState_8u** ppGITState);
```

Parameters

<i>maxSrcLen</i>	Maximum length of the source buffer.
<i>maxDstLen</i>	Maximum length of the destination buffer.
<i>ppGITState</i>	Double pointer to the GIT decoding state structure.

Description

The function `ippsDecodeGITInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the GIT decoding state structure *ppGITState*. This structure is used by the function [ippsDecodeGIT](#).

The size of the GIT decoding state structure depends on the value of the *maxDstLen* parameter, which sets the upper bound on the lengths of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>ppGITState</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>maxSrcLen</i> or <i>maxDstLen</i> is less than or equal to 0.

DecodeGITInit

Initializes the GIT decoding state structure.

Syntax

```
IppStatus ippsDecodeGITInit_8u (int maxDstLen, IppGITState_8u* pGITState);
```

Parameters

<i>maxDstLen</i>	Maximum length of the destination buffer.
<i>pGITState</i>	Pointer to memory allocated for the GIT decoding state structure.

Description

The function `ippsDecodeGITInit` is declared in the `ippdc.h` file. This function initializes the GIT decoding state structure *pGITState* in the external buffer. The size of this buffer should be determined by the function [ippsEncodeGITGetSize](#). Alternatively this structure can be initialized by the function [ippsDecodeGITInitAlloc](#).

The size of the GIT decoding state structure depends on the value of the *maxDstLen* parameter, which sets the upper bound on the lengths of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.



NOTE. The *maxDstLen* parameter of the `ippsDecodeGITInit` and [ippsDecodeGITGetSize](#) functions must be the same to avoid an error in the function [ippsDecodeGIT](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>pGITState</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>maxDstLen</i> is less than or equal to 0.

DecodeGITGetSize

Computes the size of the GIT decoding state structure.

Syntax

```
IppStatus ippsDecodeGITGetSize_8u (int maxSrcLen, int* pGITStateSize);
```

Parameters

<i>maxSrcLen</i>	Maximum length of the source buffer.
<i>pGITStateSize</i>	Pointer to the size of the GIT decoding state structure.

Description

The function `ippsDecodeGITGetSize` is declared in the `ippdc.h` file. This function computes the size in bytes *pGITStateSize* of the GIT decoding state structure. Its value depends on the *maxSrcLen* parameter, which is the maximum length of the input data buffer.

The function should be called prior to the function [ippsDecodeGITInit](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>pGITStateSize</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>maxSrcLen</i> is less than or equal to 0.

DecodeGIT

Performs GIT decoding.

Syntax

```
IppStatus ippsDecodeGIT_8u (const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int* pDstLen, IppGITStrategyHint strategyHint, IppGITState_8u* pGITState);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
-------------	-------------------------------

<i>srcLen</i>	Length of the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>strategyHint</i>	Suggests using of the code implemented specific strategy for lexicographical reordering (see Table 13-10); default value is <code>ippGITNoStrategyHint</code> .
<i>pGITState</i>	Pointer to the GIT decoding state structure.

Description

The function `ippsDecodeGIT` is declared in the `ippdc.h` file. This function performs GIT decoding - it decodes data encoded by the [ippsEncodeGIT](#) function.

The `ippsDecodeGIT` uses the GIT state structure *pGITState* that contains pointers to different data structures required for GIT decoding. This structure must be initialized by the functions [ippsDecodeGITInitAlloc](#) or [ippsDecodeGITInit](#) beforehand.

The function `ippsDecodeGIT` assumes that there is enough memory for decoding. It obtains the source data from the *pSrc* source vector with the *srcLen* length and writes the decoded data to the *pDst* destination vector. The size of destination buffer must be equal to or greater than *maxDstLen* (see [ippsDecodeGITInitAlloc](#) and [ippsDecodeGITInit](#)). *pDstLen* is an output parameter, to which the function writes the actual size of the decoded data.

The code [Example 13-7](#) shows how to use the `ippsDecodeGIT_8u` and supporting functions.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if there is not enough memory allocated for the destination buffer.

Example 13-7 Using the Function `DecodeGIT` and Supporting Functions

```
#define BLOCKSIZE 65536
.....
FILE* in, out;
Ipp8u* rdata, code;
int GITStateSize, size, codeLen, rdataLen;
IppGITState_8u pGITState;
IppGITStrategyHint strategyHint;
.....

/*****
/* Opening the file containing input compressed data and the */
/* file for the restored (uncompressed) data:                */
*****/
```

Example 13-7 Using the Function `DecodeGIT` and Supporting Functions

```
in = fopen("codefile", "rb");
out = fopen("datafile_restored.txt", "wb");

/*****
/* Memory allocation for input code and output restored data */
/* buffers, and for the internal decoding state structure: */
*****/
rdata = (Ipp8u*) malloc((BLOCKSIZE << 1) * sizeof(Ipp8u));
code = (Ipp8u*) malloc(BLOCKSIZE * sizeof(Ipp8u));
ippsDecodeGITGetSize_8u((BLOCKSIZE << 1), &GITStateSize);
pGITState = (IppGITState_8u*) malloc(GITStateSize * sizeof(Ipp8u));

/*****
/* Initializing the memory, allocated for internal decoding */
/* state structure and setting the decoding strategy */
/* (should be the same as the strategy for encoding) : */
*****/
ippsDecodeGITInit_8u(BLOCKSIZE, pGITState);
strategyHint = ippsGITLeftReorder;

/*****
/* The main loop. On every iteration program reads the size */
/* of the compressed block (codeLen) from the input code */
/* file. After that program reads the codeLen bytes of code */
/* from the input code file to the code array. The call of */
/* ippsDecodeGIT_8u performs the decoding of codeLen bytes */
/* of code to the rdata error of rdataLen. At the end */
/* program writes the rdataLen bytes of restored data(rdata) */
/* to the output file. */
*****/
for( ; ; )
{
    size = fread(&codeLen, sizeof(int), 1, in);
    if(size <= 0)
        break;
    fread(&code[0], sizeof(Ipp8u), codeLen, in);
    ippsDecodeGIT_8u(code, codeLen, rdata, &rdataLen, strategyHint,
pGITState);
    fwrite(rdata, sizeof(Ipp8u), rdataLen, out);
} /* for */
```

Example 13-7 Using the Function DecodeGIT and Supporting Functions

```

/*****
/* Free the memory allocated for input code and output */
/* restored data buffers and for the internal decoding state */
/* structure: */
*****/
free(pGITState);
free(code);
free(rdata);
fclose(in);
fclose(out);
.....

```

Move To Front Functions

This section describes the functions that performs Move To Front (MTF) data transform method. The basic idea is to represent the symbols of the source sequence as the current indexes of that symbols in the modified alphabet. This alphabet is a list where frequently used symbols are placed in the upper lines. When the given symbols occurs it is replaced by its index in the list, then this symbol is moved in the first position in the list, and all indexes are updated. For example, the sequence “baabbffffacczzdd” contains symbols that form the ordered ‘alphabet’{ 'a' , 'b' , 'c' , 'd' , 'f' , 'z' }. The function will operate in the following manner:

Table 13-11

source	destination	alphabet
		0, 1, 2, 3, 4, 5, 6
		'a', 'b', 'c', 'd', 'f', 'z'
b	1	'b', 'a', 'c', 'd', 'f', 'z'
a	1	'a', 'b', 'c', 'd', 'f', 'z'
a	0	'a', 'b', 'c', 'd', 'f', 'z'
b	1	'b', 'a', 'c', 'd', 'f', 'z'
f	4	'f', 'b', 'a', 'c', 'd', 'z'
f	0	'f', 'b', 'a', 'c', 'd', 'z'

Table 13-11

source	destination	alphabet
f	0	'f', 'b', 'a', 'c', 'd', 'z'
a	2	'a', 'f', 'b', 'c', 'd', 'z'
c	3	'c', 'a', 'f', 'b', 'd', 'z'
c	0	'c', 'a', 'f', 'b', 'd', 'z'
z	5	'z', 'c', 'a', 'f', 'b', 'd'
z	0	'z', 'c', 'a', 'f', 'b', 'd'
d	5	'd', 'z', 'c', 'a', 'f', 'b'
d	0	'd', 'z', 'c', 'a', 'f', 'b'

Finally, the function returns the destination sequence: 11014002305050.

These transformed data can be used for the following effective compression. This method is often used after Burrows-Wheeler transform.

MTFInitAlloc

Allocates memory and initializes the MTF structure.

Syntax

```
IppStatus ippsMTFInitAlloc_8u(IppMTFState_8u** ppMTFState);
```

Parameters

ppMTFState Double pointer to the MTF transform structure.

Description

The function `ippsMTFInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the MTF structure that contains required parameters for the MTF transform. This structure is used by the functions [ippsMTFFwd](#) and [ippsMTFInv](#).

Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error if *ppMTFState* pointer is NULL.

`ippMemAllocErr` Indicates an error if no memory is allocated.

MTFFree

Frees memory allocated for the MTF structure.

Syntax

```
void ippMTFFree_8u(IppMTFState_8u* pMTFState);
```

Parameters

pMTFState Pointer to the MTF structure.

Description

The function `ippMTFFree` is declared in the `ippdc.h` file. This function frees memory allocated for the MTF structure by the function [ippMTFInitAlloc](#).

MTFInit

Initializes the MTF structure.

Syntax

```
IppStatus ippMTFInit_8u(IppMTFState_8u* pMTFState);
```

Parameters

pMTFState Pointer to the MTF structure.

Description

The function `ippMTFInit` is declared in the `ippdc.h` file. This function initializes the MTF structure that contains parameters for the MTF transform in the external buffer. This structure is used by the functions [ippMTFFwd](#) and [ippMTFInv](#). The size of this buffer should be computed previously by calling the function [ippMTFGetSize](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pMTFState</i> pointer is NULL.

MTFGetSize

Computes the size of the MTF structure.

Syntax

```
IppStatus ippMTFGetSize_8u(int* pMTFStateSize);
```

Parameters

<i>pMTFStateSize</i>	Pointer to the computed MTF structure size.
----------------------	---

Description

The function `ippMTFGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) that is required for the MTF structure. Call this function prior to the function [ippMTFInit](#).

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pMTFStateSize</i> pointer is NULL.

MTFFwd

Performs the forward MTF transform.

Syntax

```
IppStatus ippMTFFwd_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, IppMTFState_8u* pMTFState);
```

Parameters

<i>pSrc</i>	Pointer to the source bufferr.
<i>pDst</i>	Pointer to the destination bufferr.
<i>srcLen</i>	Number of elements in the source and destination buffers.
<i>pMTFState</i>	Pointer to the MTF structure.

Description

The function `ippsMTFFwd` is declared in the `ippdc.h` file. This function performs the forward MTF transform of *len* elements of the data in the source buffer *pSrc* and stores result in the buffer *pDst*. The parameters of the MTF transform are specified in the MTF structure *pMTFState* that should be initialized by the functions [ippsMTFInitAlloc](#) or [ippsMTFInit](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.

MTFInv

Performs the inverse MTF transform.

Syntax

```
IppStatus ippsMTFInv_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,  
    IppMTFState_8u* pMTFState);
```

Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>len</i>	Number of elements in the source and destination buffers.
<i>pMTFState</i>	Pointer to the MTF structure.

Description

The function `ippsMTFInv` is declared in the `ippdc.h` file. This function performs the inverse MTF transform of *len* elements of data in the source buffer *pSrc* and stores result in the buffer *pDst*. The parameters of the MTF transform are specified in the MTF structure *pMTFState* that should be initialized by the functions [ippsMTFInitAlloc](#) or [ippsMTFInit](#) beforehand.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.

Run Length Encoding Functions

This section describes functions that performs data compression using Run Length Encoding (RLE) method. The main idea of this method is to replace the sequence of the same symbol by a number of its occurrences. For example, the string “aaabbbbbaaaaabbabbbbbb” will be transformed to “aa1bb2aa3bb0abb4”. The important parameters of the RLE method is a threshold - that is a minimum number of the repeated symbols exceeding of which starts up the encoding procedure. In the Intel IPP data compression functions this threshold is set to 2. Upper bound of the number of repetitions is 255 - to use one byte for repetition counter.

EncodeRLE

Performs RLE encoding.

Syntax

```
IppStatus ippsEncodeRLE_8u(const Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst, int* pDstLen);
```

Parameters

<i>ppSrc</i>	Double pointer to the source data buffer.
<i>pSrcLen</i>	Pointer to the number of elements in the source buffer after encoding point to the number of remaining elements.
<i>pDst</i>	Pointer to the destination data buffer.

pDstLen Pointer to the number of elements in the destination buffer, after encoding to the actual number of elements in the destination buffer.

Description

The function `ippsEncoderLE` is declared in the `ippdc.h` file. This function performs RLE encoding *pSrcLen* elements of the source data buffer *ppSrc* and stores the result in the buffer *pDst*. After encoding the function updates the *pSrcLen* thus it points to the number of remaining elements of the source buffer, and *pDstLen* points to the actual number of elements stored in the destination buffer.

Sometimes the number of output elements can exceed the specified size of destination buffer. In this case the function returns the warning message and pointer to the number of not-encoded source elements. The user can call the function `ippsEncoderLE` again to complete encoding.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is null.
<code>ippStsSizeErr</code>	Indicates an error if <i>pSrcLen</i> or <i>pDstLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

DecodeRLE

Performs RLE decoding.

Syntax

```
IppStatus ippsDecodeRLE_8u(const Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst, int* pDstLen);
```

Parameters

<i>ppSrc</i>	Double pointer to the source data buffer.
<i>pSrcLen</i>	Pointer to the number of elements in the source buffer, after decoding pointer to the number of remaining elements.

<i>pDst</i>	Pointer to the destination data buffer.
<i>pDstLen</i>	Pointer to the number of elements in the destination buffer, after encoding to the actual number of elements in the destination buffer.

Description

The function `ippSDecoderLE` is declared in the `ippdc.h` file. This function performs RLE decoding *srcLen* elements of the source data buffer *ppSrc* and stores the result in the destination buffer *pDst*. After decoding the function updates the *pSrcLen* thus it points to the number of remaining elements of the source buffer, and *pDstLen* points to the actual number of elements stored in the destination buffer.

Sometimes the number of output elements can exceed the specified size of destination buffer. In this case the function returns the warning message and pointer to the number of not-decoded source elements. The user can call the function `ippSDecoderLE` again to complete decoding.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>pSrcLen</i> or <i>pDstLen</i> is less than or equal to 0.
<code>ippStsSrcDataErr</code>	Indicates an error if <i>ppSrc</i> contains unsupported data.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

Handling of Special Cases

A

Some mathematical functions implemented in Intel IPP are not defined for all possible argument values. This appendix describes how the corresponding Intel IPP functions used in signal processing data-domain handle situations when their input arguments fall outside the range of function definition or may lead to ambiguously determined output results.

[Table A-1](#) below summarizes these special cases for general vector functions described in chapter 5 and lists result values together with status codes returned by these functions.

Table A-1 Special Cases for Intel IPP Signal Processing Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsSqrt	16s	Sqrt ($x < 0$)	0	ippStsSqrtNegArg
	32f	Sqrt ($x < 0$)	NAN_32F	ippStsSqrtNegArg
	64s	Sqrt ($x < 0$)	0	ippStsSqrtNegArg
	64f	Sqrt ($x < 0$)	NAN_64F	ippStsSqrtNegArg
ippsDiv	8u	Div (0/0)	0	ippStsDivByZero
		Div ($x/0$)	IPP_MAX_8U	ippStsDivByZero
	16s	Div (0/0)	0	ippStsDivByZero
		Div ($x/0$), $x > 0$	IPP_MAX_16S	ippStsDivByZero
		Div ($x/0$), $x < 0$	IPP_MIN_16S	ippStsDivByZero
	16sc	Div (0/0)	0	ippStsDivByZero
		Div ($x/0$)	0	ippStsDivByZero

Table A-1 Special Cases for Intel IPP Signal Processing Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsDiv	32f	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0), x>0	INF_32F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_32F	ippStsDivByZero
	32fc	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0)	NAN_32F	ippStsDivByZero
				ippStsDivByZero
	64f	Div (0/0)	NAN_64F	
		Div (x/0), x>0	INF_64F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_64F	ippStsDivByZero
	64fc	Div (0/0)	NAN_64F	ippStsDivByZero
		Div (x/0)	NAN_64F	ippStsDivByZero
ippsDivC	All	Div(x/0)	x	ippStsDivByZeroErr
ippsLn	16s	Ln (0)	IPP_MIN_16S	ippStsLnZeroArg
		Ln (x<0)	IPP_MIN_16S	ippStsLnNegArg
	32s	Ln (0)	IPP_MIN_32S	ippStsLnZeroArg
		Ln (x<0)	IPP_MIN_32S	ippStsLnNegArg
	32f	Ln (x<0)	NAN_32F	ippStsLnNegArg
		Ln(x<IPP_MINABS_32F)	INF_NEG_32F	ippStsLnZeroArg
	64f	Ln (x<0)	NAN_64F	ippStsLnNegArg
		Ln(x<IPP_MINABS_64F)	INF_NEG_64F	ippStsLnZeroArg
ippsExp	16s	overflow	IPP_MAX_16S	ippStsNoErr
	32s	overflow	IPP_MAX_32S	ippStsNoErr
	64s	overflow	IPP_MAX_64S	ippStsNoErr
	32f	overflow	INF_32F	ippStsNoErr
	64f	overflow	INF_64F	ippStsNoErr

Table A-1 **Special Cases for Intel IPP Signal Processing Functions** (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsthreshold ,	16sc	level<0	x	ippStsThreshNegLevelErr
ippsthresholdLT ,	32fc	level<0	x	ippStsThreshNegLevelErr
ippsthresholdGT ,	64fc	level<0	x	ippStsThreshNegLevelErr
ippsthresholdLTval ,				
ippsthresholdGTval				
ippsthresholdLTinv	32f	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_32F	ippStsInvZero
	32fc	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_32F	ippStsInvZero
	64f	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_64F	ippStsInvZero
	64fc	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_64F	ippStsInvZero
ippsl0log10	32s	10Log10 (0)	IPP_MIN_32S	ippStsLnZeroArg
		10Log10 (x<0)	IPP_MIN_32S	ippStsLnNegArg

Here x denotes an input value. For the definition of the constants used, see [“Data Ranges”](#) in Chapter 2.

Note that flavors of the same math function operating on different data types may produce unlike results for the equal argument values. However, for a given function and a fixed data type, handling of special cases is the same for all function flavors that have different [descriptors](#) in their names. For example, logarithm function `ippsLn` operating on `16s` data treats zero argument values in the same way for all its flavors `ippsLn_16s_Sfs` and `ippsLn_16s_ISfs`.

[Table A-2](#) below summarizes special cases for fixed accuracy arithmetic vector functions described in the Chapter 12 [“Fixed-Accuracy Arithmetic Functions”](#).

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
<u>ippsInv</u>	32f	Inv (x ==+0)	INF_32F	ippStsSingularity
		Inv (x ==-0)	-INF_32F	ippStsSingularity
	64f	Inv (x ==+0)	INF_64F	ippStsSingularity
		Inv (x ==-0)	-INF_64F	ippStsSingularity
<u>ippsDiv</u>	32f	Div (x>0, y==+0)	INF_32F	ippStsSingularity
		Div (x>0, y=-0)	-INF_32F	ippStsSingularity
		Dlv (x<0, y==+0)	-INF_32F	ippStsSingularity
		Div (x<0, y=-0)	INF_32F	ippStsSingularity
		Div (x=0, y=0)	NAN_32F	ippStsSingularity
	64f	Div (x>0, y==+0)	INF_64F	ippStsSingularity
		Div (x>0, y=-0)	-INF_64F	ippStsSingularity
		Dlv (x<0, y==+0)	-INF_64F	ippStsSingularity
		Div (x<0, y=-0)	INF_64F	ippStsSingularity
		Div (x=0, y=0)	NAN_64F	ippStsSingularity
<u>ippsSqrt</u>	32f	Sqrt (x<0)	NAN_32F	ippStsDomain
		Sqrt (x=-INF)	NAN_32F	ippStsDomain
	64f	Sqrt (x<0)	NAN_64F	ippStsDomain
		Sqrt (x=-INF)	NAN_64F	ippStsDomain
<u>ippsInvSqrt</u>	32f	InvSqrt (x<0)	NAN_32F	ippStsDomain
		InvSqrt (x==+0)	INF_32F	ippStsSingularity
		InvSqrt (x=-0)	-INF_32F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_32F	ippStsDomain
	64f	InvSqrt (x<0)	NAN_64F	ippStsDomain
		InvSqrt (x==+0)	INF_64F	ippStsSingularity
		InvSqrt (x=-0)	-INF_64F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_64F	ippStsDomain
<u>ippsInvCbrt</u>	32f	InvCbrt (x==+0)	INF_32F	ippStsSingularity
		InvCbrt (x=-0)	-INF_32F	ippStsSingularity
	64f	InvCbrt (x==+0)	INF_64F	ippStsSingularity
		InvCbrt (x=-0)	-INF_64F	ippStsSingularity

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
<u>ippsPow</u>	32f	Pow (x=+0, y=-ODD_INT)	INF_32F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_32F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_32F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_32F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_32F	ippStsDomain
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_64F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_64F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_64F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_64F	ippStsDomain
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity
<u>ippsPowx</u>	32f	Pow (x=+0, y=-ODD_INT)	INF_32F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_32F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_32F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_32F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_32F	ippStsDomain
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_64F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_64F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_64F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_64F	ippStsDomain
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity
<u>ippsExp</u>	32f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_32F	ippStsOverflow
	64f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_64F	ippStsOverflow

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
<u>ippsLn</u>	32f	Ln ($x < 0$)	NAN_32F	ippStsDomain
		Ln ($x = -\text{INF}$)	NAN_32F	ippStsDomain
		Ln ($x = 0$)	-INF_32F	ippStsSingularity
	64f	Ln ($x < 0$)	NAN_64F	ippStsDomain
		Ln ($x = -\text{INF}$)	NAN_64F	ippStsDomain
		Ln ($x = 0$)	-INF_64F	ippStsSingularity
<u>ippsLog10</u>	32f	Ln ($x < 0$)	NAN_32F	ippStsDomain
		Ln ($x = -\text{INF}$)	NAN_32F	ippStsDomain
		Ln ($x = 0$)	-INF_32F	ippStsSingularity
	64f	Ln ($x < 0$)	NAN_64F	ippStsDomain
		Ln ($x = -\text{INF}$)	NAN_64F	ippStsDomain
		Ln ($x = 0$)	-INF_64F	ippStsSingularity
<u>ippsCos</u>	32f	Cos (INF)	NAN_32F	ippStsDomain
	64f	Cos (INF)	NAN_64F	ippStsDomain
<u>ippsSin</u>	32f	Sin (INF)	NAN_32F	ippStsDomain
	64f	Sin (INF)	NAN_64F	ippStsDomain
<u>ippsSinCos</u>	32f	SinCos (INF)	NAN_32F, NAN_32F	ippStsDomain
	64f	SinCos (INF)	NAN_64F, NAN_64F	ippStsDomain
<u>ippsTan</u>	32f	Tan (INF)	NAN_32F	ippStsDomain
	64f	Tan (INF)	NAN_64F	ippStsDomain
<u>ippsAcos</u>	32f	Acos (x), $ x > 1$	NAN_32F	ippStsDomain
		Acos (INF)	NAN_32F	ippStsDomain
	64f	Acos (x), $ x > 1$	NAN_64F	ippStsDomain
		Acos (INF)	NAN_64F	ippStsDomain
<u>ippsAsin</u>	32f	Acos (x), $ x > 1$	NAN_32F	ippStsDomain
		Acos (INF)	NAN_32F	ippStsDomain
	64f	Acos (x), $ x > 1$	NAN_64F	ippStsDomain
		Acos (INF)	NAN_64F	ippStsDomain
<u>ippsCosh</u>	32f	Cosh (x), $ x > \text{overflow}$	INF_32F	ippStsOverflow
	64f	Cosh (x), $ x > \text{overflow}$	INF_64F	ippStsOverflow
<u>ippsSinh</u>	32f	Sinh (x), $ x > \text{overflow}$	INF_32F	ippStsOverflow
	64f	Sinh (x), $ x > \text{overflow}$	INF_64F	ippStsOverflow

Table A-2 **Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions** (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsAcosh	32f	Acosh ($x < 1$)	NAN_32F	ippStsDomain
		Acosh ($x = -\text{INF}$)	NAN_32F	ippStsDomain
	64f	Acosh ($x < 1$)	NAN_64F	ippStsDomain
		Acosh ($x = -\text{INF}$)	NAN_64F	ippStsDomain
ippsAtanh	32f	Atanh ($x = 1$)	INF_32F	ippStsSingularity
		Atanh ($x = -1$)	-INF_32F	ippStsSingularity
		Atanh (x), $ x > 1$	NAN_32F	ippStsDomain
		Atanh (INF)	NAN_32F	ippStsDomain
	64f	Atanh ($x = 1$)	INF_64F	ippStsSingularity
		Atanh ($x = -1$)	-INF_64F	ippStsSingularity
		Atanh (x), $ x > 1$	NAN_64F	ippStsDomain
		Atanh (INF)	NAN_64F	ippStsDomain
ippsErfc	32f	Erfc (x), $x < \text{underflow}$	0	ippStsUnderflow
	64f	Exp (x), $x < \text{underflow}$	0	ippStsUnderflow

Removed Functions



This appendix contains [Table B-1](#) that lists the functions that have been removed from the Intel IPP version 5.0. If an application created with the previous versions calls function listed here, then the source code must be modified. The table specifies the corresponding Intel IPP 5.0 functions to substitute for removed functions.

Table B-1 Removed Functions

Removed from 5.0	Substitution or Workaround	Comments
Audio Coding Fucntions		
ippsApplySF_32f_I	ippsScale_32f_I	
ippsBuildHDT_32s	ippsVLCDecodeInit_32s	Data Compression domain
ippsBuildHET_16s	ippsVLCEncodeInit_32s	Data Compression domain
ippsBuildHET_VLC_32s	ippsVLCEncodeInit_32s	Data Compression domain
ippsCdbkFree_VQ_32f	ippsVQCodeBookFree_32f	
ippsCdbkInitAlloc_VQ_32f	ippsVQCodeBookInitAlloc_32f	
ippsCountBits_1tuple_VLC_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsCountBits_2tuple_VLC_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsCountBits_4tuple_VLC_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsDecodeVLC_32s	ippsVLCDecodeOne_1u16s	Data Compression domain
ippsDecodeVLC_AACESCBBlock_32s	ippsVLCDecodeEscBlock_AAC_1u16s	
ippsDecodeVLC_Block_32s	ippsVLCDecodeBlock_1u16s	Data Compression domain
ippsDecodeVLC_MP3ESCBBlock_32s	ippsVLCDecodeEscBlock_MP3_1u16s	
ippsEncodeBlock_1tuple_VLC_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeBlock_2tuple_VLC_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeBlock_4tuple_VLC_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain

Table B-1 Removed Functions

Removed from 5.0	Substitution or Workaround	Comments
ippsEncodeVLC_Block_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeVLC_MP3ESCBBlock_16s	ippsVLCEncodeEscBlock_MP3_16s1u	
ippsFDPIInv_32f	ippsFDPIInv_32f_I	
ippsGetSizeHDT_32s	ippsVLCDecodeGetSize_32s	Data Compression domain
ippsGetSizeHET_16s	ippsVLCEncodeGetSize_32s	Data Compression domain
ippsGetSizeHET_VLC_32s	ippsVLCEncodeGetSize_32s	Data Compression domain
ippsHuffmanCountBits_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsIndexSelect_VQ_32f	ippsVQIndexSelect_32f	
ippsMainSelect_VQ_32f	ippsVQMainSelect_32f	
ippsPreSelect_VQ_32f	ippsVQPreliminarySelect_32f	
ippsResetFDP_32f	ippsFDPReset_32f	
ippsResetFDP_SFB_32f	ippsFDPResetSfb_32f	
ippsResetFDPGroup_32f	ippsFDPResetGroup_32f	
ippsVectorReconstruction_VQ_32f	ippsVQReconstruction_32f	
Common Fucntions		
ippCoreGetCpuClocks	ippGetCpuClocks	
ippCoreGetCpuType	ippGetCpuType	
ippCoreGetStatusString	ippGetStatusString	
ippCoreSetDenormAreZeros	ippSetDenormAreZeros	
ippCoreSetFlushToZero	ippSetFlushToZero	
ippStaticFree		useless, no memory allocation
ippStaticInitBest	ippStaticInit	
Speech Recognition Fucntions		
ippsAddMulCol_64f_D2L	ippsAddMulColumn_64f_D2L	
ippsAddNRow_32f_D2	ippsAddNRows_32f_D2	
ippsCoefUpdateAECNLMS_32fc_I		
ippsCoefUpdateAECNLMS_32sc_I		supported in the current version
ippsControllerGetSizeAEC_32f		
ippsControllerGetSizeAEC_32s		supported in the current version
ippsControllerInitAEC_32f		
ippsControllerInitAEC_32s		supported in the current version
ippsControllerUpdateAEC_32f		
ippsControllerUpdateAEC_32s		supported in the current version

Table B-1 Removed Functions

Removed from 5.0	Substitution or Workaround	Comments
ippsCopyCol_32f_D2	ippsCopyColumn_32f_D2	
ippsDeltaDeltaW1_32f_D2	ippsDeltaDelta_Win1_32f_D2	
ippsDeltaDeltaW2_32f_D2	ippsDeltaDelta_Win2_32f_D2	
ippsDeltaW1_32f_D2	ippsDelta_Win1_32f_D2	
ippsDeltaW2_32f_D2	ippsDelta_Win2_32f_D2	
ippsDotProdCol_64f_D2L	ippsDotProdColumn_64f_D2L	
ippsFilterAECNLMS_32fc		
ippsFilterAECNLMS_32sc_Sfs		supported in the current version
ippsLMThreshold_32f	ippsScaleLM_32f	
ippsLogGauss_16s32f_D2	ippsLogGauss_Scaled_16s32f_D2	
ippsLogGauss_16s32f_D2L	ippsLogGauss_Scaled_16s32f_D2L	
ippsLogGauss_IdVar_16s32f_D2	ippsLogGauss_IdVarScaled_16s32f_D2	
ippsLogGauss_IdVar_16s32f_D2L	ippsLogGauss_IdVarScaled_16s32f_D2L	
ippsLogGauss_IdVarLow_16s32f_D2	ippsLogGauss_IdVarLowScaled_16s32f_D2	
ippsLogGauss_IdVarLow_16s32f_D2L	ippsLogGauss_IdVarLowScaled_16s32f_D2L	
ippsLogGauss_Low_16s32f_D2	ippsLogGauss_LowScaled_16s32f_D2	
ippsLogGauss_Low_16s32f_D2L	ippsLogGauss_LowScaled_16s32f_D2L	
ippsLogGauss1_32f_D2	ippsLogGauss_32f_D2	
ippsLogGauss1_32f_D2L	ippsLogGauss_32f_D2L	
ippsLogGauss1_64f_D2	ippsLogGauss_64f_D2	
ippsLogGauss1_64f_D2L	ippsLogGauss_64f_D2L	
ippsLogGauss2_32f_D2	ippsLogGaussAddMultiMix_32f_D2	
ippsLogGauss2_32f_D2L	ippsLogGaussAddMultiMix_32f_D2L	
ippsLogGauss2_64f_D2	ippsLogGaussAddMultiMix_64f_D2	
ippsLogGauss2_64f_D2L	ippsLogGaussAddMultiMix_64f_D2L	
ippsLogGaussAdd_16s32f_D2	ippsLogGaussAdd_Scaled_16s32f_D2	
ippsLogGaussAdd_16s32f_D2L	ippsLogGaussAdd_Scaled_16s32f_D2L	

Table B-1 **Removed Functions**

Removed from 5.0	Substitution or Workaround	Comments
ippsLogGaussAdd_IdVar_16s32f_D2	ippsLogGaussAdd_IdVarScaled_16s32f_D2	
ippsLogGaussAdd_IdVar_16s32f_D2L	ippsLogGaussAdd_IdVarScaled_16s32f_D2L	
ippsLogGaussAdd1_32f_D2	ippsLogGaussAdd_32f_D2	
ippsLogGaussAdd2_32f_D2	ippsLogGaussAddMultiMix_32f_D2	
ippsLogGaussAddMultiMix_16s32f_D2	ippsLogGaussAddMultiMix_Scaled_16s32f_D2	
ippsLogGaussAddMultiMix_16s32f_D2L	ippsLogGaussAddMultiMix_Scaled_16s32f_D2L	
ippsLogGaussMax_16s32f_D2	ippsLogGaussMax_Scaled_16s32f_D2	
ippsLogGaussMax_16s32f_D2L	ippsLogGaussMax_Scaled_16s32f_D2L	
ippsLogGaussMax_IdVar_16s32f_D2	ippsLogGaussMax_IdVarScaled_16s32f_D2	
ippsLogGaussMax_IdVar_16s32f_D2L	ippsLogGaussMax_IdVarScaled_16s32f_D2L	
ippsLogGaussMax1_32f_D2	ippsLogGaussMax_32f_D2	
ippsLogGaussMaxMultiMix_16s32f_D2	ippsLogGaussMaxMultiMix_Scaled_16s32f_D2	
ippsLogGaussMaxMultiMix_16s32f_D2L	ippsLogGaussMaxMultiMix_Scaled_16s32f_D2L	
ippsLogGaussMultiMix_16s32f_D2	ippsLogGaussMultiMix_Scaled_16s32f_D2	
ippsLogGaussMultiMix_16s32f_D2L	ippsLogGaussMultiMix_Scaled_16s32f_D2L	
ippsLogGaussMultiMix_Low_16s32f_D2	ippsLogGaussMultiMix_LowScaled_16s32f_D2	
ippsLogGaussMultiMix_Low_16s32f_D2L	ippsLogGaussMultiMix_LowScaled_16s32f_D2L	
ippsLogGaussSingle_16s32f	ippsLogGaussSingle_Scaled_16s32f	
ippsLogGaussSingle_BlockDVar_16s32f	ippsLogGaussSingle_BlockDVarScaled_16s32f	
ippsLogGaussSingle_DirectVar_16s32f	ippsLogGaussSingle_DirectVarScaled_16s32f	

Table B-1 Removed Functions

Removed from 5.0	Substitution or Workaround	Comments
ippsLogGaussSingle_IdVar_16s32f	ippsLogGaussSingle_IdVarScaled_16s32f	
ippsLogGaussSingle_IdVarLow_16s32f	ippsLogGaussSingle_IdVarLowScaled_16s32f	
ippsLogGaussSingle_Low_16s32f	ippsLogGaussSingle_LowScaled_16s32f	
ippsLPToLSP_16s_D2Sfs	ippsLPToLSP_16s_Sfs	
ippsLPToLSP_32f_D2	ippsLPToLSP_32f	
ippsLSPToLP_16s_D2Sfs	ippsLSPToLP_16s_Sfs	
ippsLSPToLP_32f_D2	ippsLSPToLP_32f	
ippsMahDist1_32f_D2	ippsMahDist_32f_D2	
ippsMahDist1_32f_D2L	ippsMahDist_32f_D2L	
ippsMahDist1_64f_D2	ippsMahDist_64f_D2	
ippsMahDist1_64f_D2L	ippsMahDist_64f_D2L	
ippsMahDist2_32f_D2	ippsMahDistMultiMix_32f_D2	
ippsMahDist2_32f_D2L	ippsMahDistMultiMix_32f_D2L	
ippsMahDist2_64f_D2	ippsMahDistMultiMix_64f_D2	
ippsMahDist2_64f_D2L	ippsMahDistMultiMix_64f_D2L	
ippsMeanCol_32f_D2	ippsMeanColumn_32f_D2	
ippsMeanCol_32f_D2L	ippsMeanColumn_32f_D2L	
ippsMeanVarCol_32f_D2	ippsVarColumn_32f_D2	
ippsMeanVarCol_32f_D2L	ippsVarColumn_32f_D2L	
ippsMulCol_64f_D2L	ippsMulColumn_64f_D2L	
ippsNormalizeCol_32f_D2	ippsNormalizeColumn_32f_D2	
ippsQRTransCol_64f_D2L	ippsQRTransColumn_64f_D2L	
ippsRecSqrt_32f_Th	ippsRecSqrt_32f	
ippsStepSizeUpdateAECNLMS_32f		supported in the current version
ippsStepSizeUpdateAECNLMS_32s		
ippsSumAllRow_32f_D2	ippsSumRow_32f_D2	
ippsSumAllRow_32f_D2L	ippsSumRow_32f_D2L	
ippsSumCol_32f_D2	ippsSumColumn_32f_D2	
ippsSumCol_32f_D2L	ippsSumColumn_32f_D2L	
ippsSumCol_64f_D2	ippsSumColumn_64f_D2	
ippsSumCol_64f_D2L	ippsSumColumn_64f_D2L	

Table B-1 **Removed Functions**

Removed from 5.0	Substitution or Workaround	Comments
ippsSumColAbs_64f_D2L	ippsSumColumnAbs_64f_D2L	
ippsSumColSqr_64f_D2L	ippsSumColumnSqr_64f_D2L	
ippsVarCol_32f_D2	ippsVarColumn_32f_D2	
ippsVarCol_32f_D2L	ippsVarColumn_32f_D2L	

Bibliography

This bibliography provides a list of reference books and other sources of additional information that might be useful to the application programmer. This list is neither complete nor exhaustive, but serves as a starting point. Of all the references listed, [Mit93] will be the most useful to those readers who already have a basic understanding of signal processing. This reference collects the work of 27 experts in the field and has both great breadth and depth.

The books [Opp75], [Opp89], [Jac89], and [Zie83] are undergraduate signal processing texts. [Opp89] is a much revised edition of the classic [Opp75]; [Jac89] is more concise than the others; and [Zie83] also covers continuous-time systems.

- [Arn97] Z. Arnavut, S. Magliveras. *Block-Sorting and Compression*. IEEE Data Compression Conference, Snowbird, Utah, pp.181-190, March 1997.
- [Ash94] M.R. Asharif and F. Amano. *Acoustic Echo Canceled Using the FBAF Algorithm*. IEEE Trans. Comm., Vol. 42, No. 12, pp. 3090-3094, Dec. 1994.
- [Brislawn94] C. Brislawn, *Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks*. Los Alamos Report LA-UR-94-1747, 1994.
- [Cap78] V. Cappellini, A. G. Constantinides, and P. Emilani. *Digital Filters and Their Applications*. Academic Press, London, 1978.
- [Cap94] Cappé O., *Elimination of the musical noise phenomenon with the Ephraim and Malah noise suppressor*. IEEE Trans. Speech and Audio Processing, vol. 2(2), (1994).
- [CCITT] CCITT, Recommendation G.711. *Pulse Code Modulation of Frequencies*, 1984.
- [Coh02] I. Cohen and B. Berdugo. *Noise Estimation by Minima Controlled Recursive Averaging for Robust Speech Enhancement*. IEEE Signal Proc. Letters, Vol. 9, No. 1, Jan. 2002, pp. 12-15.

- [Cro83] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1983.
- [Dau92] I. Daubechies. *Ten Lectures on Wavelets*. Springer Verlag, Pennsylvania, 1992.
- [Eph84] Y. Ephraim and D. Malah. *Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator*. IEEE Trans. ASSP, Vol. 32, No. 6, Dec. 1984, pp. 1109-1121.
- [ES201] ETSI ES 201 108 V1.1.2. ETSI Standard. *Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Front-end feature extraction algorithm; Compression algorithms*.
- [ES202] ETSI ES 202 050 V1.1.1. ETSI Standard. *Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Advanced front-end feature extraction algorithm; Compression algorithms*.
- [Feig92] E. Feig and S. Winograd. *Fast algorithms for DCT*. IEEE Transactions on Signal Processing, vol.40, No.9, 1992.
- [Flet82] J.G.Fletcher. *An Arithmetic Checksum for Serial Transmissions*. IEEE Transactions on Communications, vol.COM-30, No.1, 1982, pp.247-252.
- [Gal75] R. Gallager, D.Van Voorhis. *Optimal source codes for geometrically distributed integer alphabets*. IEEE Transactions on Information Theory, vol. IT-21, No. 3, pp.228-230, 1975.
- [Griff87] G.Griffiths, G.C.Stones. *The tea-leaf reader algorithm: an efficient implementation of CRC-16 and CRC-32*. Communications of the ACM , vol.30, No.7, 1987, pp.617-620.
- [Ham83] R.W. Hamming. *Digital Filters*, Prentice-Hall, New Jersey 1983.
- [Har78] F. Harris. *On the Use of Windows*. Proceedings of the IEEE, vol. 66, No.1, IEEE, 1978.
- [Hay91] S. Haykin. *Adaptive Filter Theory*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

- [ISO11172] ISO/IEC 11172-3 - Information technology. *Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s*. Part 3: Audio (1993)
- [ISO13818] ISO/IEC 13818-3 - Information technology. *Generic coding of moving pictures and associated audio information*. Part 3: Audio (1998).
- [ISO14496] ISO/IEC 14496-3 - Information technology. *Coding of audio-visual objects*. Part 3: Audio (2001).
- [ISO14496A] ISO/IEC 14496-3/Amd.1:2003 - Bandwidth Extension (2003).
- [ITU224] ITU-T Recommendation X.224 Annex D. *Checksum Algorithms*, (11/93), pp144,145.
- [ITU722] ITU-T Recommendation G.722.1 (09/99), *Coding at 24 and 32 kbit/s for hand-free operation in systems with low frame loss*.
- [ITU723] ITU-T Recommendation G.723.1 . *Dual Rate speech coder for Multimedia Communications transmitting at 5.3 and 6.3 Kbit/s*, (03/96).
- [ITU723A] ITU-T Recommendation G.723.1 Annex A. *Silence compression scheme*, (11/96).
- [ITU729] ITU-T Recommendation G.729. *Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)*, (03/96).
- [ITU729A] ITU-T Recommendation G.729 Annex A. *Reduced Complexity 8 kbit/s CS-ACELP speech codec*, (11/96).
- [ITU729B] ITU-T Recommendation G.729 Annex B. *A silence compression scheme for G.729 optimized for terminals conforming to Recommendation V.70*, (10/96).
- [ITU729B1] ITU-T Recommendation G.729 Annex B Corrigendum 1, (02/98).
- [ITUV34] ITU-T Recommendation V.34. *A modem operating at data signalling rates of up to 33 600 bit/s for use on the general switched telephone network and on leased point-to-point 2-wire telephone-type circuit*. (02/98).

- [Jack89] Leland B. Jackson. *Digital Filters and Signal Processing*. Kluwer Academic Publishers, second edition, 1989.
- [Kab86] P. Kabal and P.Ramachandran. *The Computation of line Spectral Frequencies Using Chebyshev Polynomials*. IEEE transaction on acoustic, speech and signal processing, vol. ASSP-34, No.6, 1986.
- [Lyn89] Paul A. Lynn. *Introductory Digital Signal Processing with Computer Applications*. John Wiley&Sons, Inc., New York, 1993.
- [Mar01] R. Martin. *Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics*. IEEE Trans. Speech and Audio, Vol. 9, No. 5, July 2001, pp. 504-512.
- [Med91] Y. Medan, E. Yair, D. Chazan. *Super Resolution Pitch Determination of Speech Signals*. IEEE Transactions on Signal Processing, vol 39, No.1, 1991.
- [Mit93] Sanjit K. Mitra and James F. Kaiser editors. *Handbook for Digital Signal Processing*. John Wiley&Sons, Inc., New York, 1993.
- [Mit98] S. K. Mitra. *Digital Signal Processing*. McGraw Hill, 1998.
- [Nel92] M.R.Nelson. *File verification using CRC 32-bit cyclical redundancy check*. Dr.Dobb's Journal, vol. 17, Issue 5, 1992, p.64.
- [NIC91] Nam Ik Cho and Sang Uk Lee. *Fast algorithm and implementation of 2D DCT*. IEEE Transactions on Circuits and Systems, vol. 31, No.3, 1991.
- [Opp75] A.V. Oppenheim and R.W. Schafer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.
- [Opp89] A.V. Oppenheim and R.W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [Rab78] L.R. Rabiner and R.W. Schafer. *Digital Processing of Speech Signals*. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

- [Rao90] K.R. Rao and P. Yip. *Discrete Cosine Transform. Algorithms, Advantages and Applications*. Academic Press, San Diego, 1990.
- [RFC1950] P.Deutsch, J-L.Gailly. *ZLIB Compressed Data Format Specification version 3.3*, May, 1996.
<http://www.ietf.org/rfc/rfc1950.txt>
- [RFC1951] P.Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*, May, 1996.
<http://www.ietf.org/rfc/rfc1951.txt>
- [RFC1952] P.Deutsch. *GZIP file format specification version 4.3*, May, 1996.
<http://www.ietf.org/rfc/rfc1952.txt>
- [Seg78] R. Sedgewick. *Implementing quicksort programs*. Communications of the ACM, Vol. 21, No. 10, pp. 847-857, Oct. 1978.
- [Storer82] J. Storer, T.Szymanski. *Data Compression via Textual Substitution*. Journal of the Association for Computing Machinery (ACM), vol.19, No.4, pp.928-951, Oct.1982.
- [Strang96] G. Strang and T. Nguyen. *Wavelet and Filter Banks*. Wellesley-Cambridge Press, 1996, pp. 153-157.
- [Tuc92] R. Tucker. *Voice Activity Detection Using a Periodicity Measure*. IEEE Proceedings-I, vol. 139, No. 4, August 1992, pp. 377-380.
- [Vai93] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, Englewood Cliffs, New Jersey.
- [Wid85] B. Widrow and S.D. Stearns. *Adaptive Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
- [Zie83] Rodger E. Ziemer, William H. Tranter and D. Ronald Fannin. *Signals and Systems: Continuous and Discrete*. Macmillan Publishing Co., New York, 1983.
- [Ziv77] J.Ziv and A.Lempel. *A Universal Algorithm for Sequential Data Compression*. IEEE Transactions on Information Theory, vol.23, No.3, pp.337-343, May 1977.

Glossary

adaptive filter	An adaptive filter varies its filter coefficients (taps) over time. Typically, the filter's coefficients are varied to make its output match a prototype "desired" signal as closely as possible. Non-adaptive filters do not vary their filter coefficients over time.
arithmetic operation	An operation that adds, subtracts, multiplies, or squares the image pixel values.
BQ	One of the modes, which indicates that the IIR initialization function initializes a cascade of biquads.
CCS	See complex conjugate-symmetric.
companding functions	The functions that perform an operation of data compression by using a logarithmic encoder-decoder. Companding allows you to maintain the percentage error constant by logarithmically spacing the quantization levels.
complex conjugate-symmetric	A kind of symmetry that arises in the Fourier transform of real signals. A complex conjugate-symmetric signal has the property that $x(-n) = x(n)^*$, where "*" denotes conjugation.
conjugate	The conjugate of a complex number $a + bj$ is $a - bj$.
conjugate-symmetric	See complex conjugate-symmetric.
DCT	Acronym for the discrete cosine transform.
decimation	Filtering a signal followed by down-sampling. Filtering prevents aliasing distortion in the subsequent down-sampling. See down-sampling.

down-sampling	Down-sampling conceptually decreases a signal's sampling rate by removing samples from between neighboring samples of a signal. See decimation.
element-wise	<p>An element-wise operation performs the same operation on each element of a vector, or uses the elements of the same position in multiple vectors as inputs to the operation. For example, the element-wise addition of the vectors $\{x_0, x_1, x_2\}$ and $\{y_0, y_1, y_2\}$ is performed as follows:</p> $\{x_0, x_1, x_2\} + \{y_0, y_1, y_2\} = \{x_0 + y_0, x_1 + y_1, x_2 + y_2\}.$
FIR	Abbreviation for finite impulse response filter. Finite impulse response filters do not vary their filter coefficients (taps) over time.
FIR LMS	Abbreviation for least mean squares finite impulse response filter.
fixed-point data format	A format that assigns one bit for a sign and all other bits for fractional part. This format is used for optimized conversion operations with signed, purely fractional vectors. For example, S.31 format assumes a sign bit and 31 fractional bits; S15.16 assumes a sign bit, 15 integer bits, and 16 fractional bits.
IIR	Abbreviation for infinite impulse response filters.
in-place	A function that performs its operation in-place, takes its input from an array and returns its output to the same array. See not-in-place.
interpolation	Up-sampling a signal followed by filtering. The filtering gives the inserted samples a value close to the samples of their neighboring samples in the original signal. See up-sampling.
LMS	Abbreviation for least mean square, an algorithm frequently used as a measure of the difference between two signals. Also used as shorthand for an adaptive FIR filter employing the LMS algorithm for adaptation.

LTI	Abbreviation for linear time-invariant systems. In LTI systems, if an input consists of the sum of a number of signals, then the output is the sum of the system's responses to each signal considered separately [Lyn89].
MMX [™] technology	An enhancement to Intel architecture aimed at better performance in multimedia and communications applications. The technology uses four additional data types, eight 64-bit MMX registers, and 57 additional instructions implementing the SIMD (single instruction, multiple data) technique.
MR	One of the modes, indicating the multi-rate variety of the function.
multi-rate	An operation or signal processing system involving signals with multiple sample rates. Decimation and interpolation are examples of multi-rate operations.
not-in-place	A function that performs its operation not-in-place takes its input from a source array and puts its output in a second, destination array.
polyphase	A computationally efficient method for multi-rate filtering. For example, interpolation or decimation.
CCS	A representation of a complex conjugate-symmetric sequence which is easier to use than the Pack or Perm formats.
Pack	A compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real FFT algorithms ("natural" in the sense that bit-reversed order is natural for radix-2 complex FFTs).
Perm	A format for storing the values for the FFT algorithm. RCPPerm format stores the values in the order in which the FFT algorithm uses them. That is, the real and imaginary parts of a given sample need not be adjacent.

saturation	Using saturation arithmetic, when a number exceeds the data-range limit for its data type, it saturates to the upper data-range limit. For example, a signed word greater than 7FFFh saturates to 7FFFh. When a number is less than the lower data-range limit, it saturates to the lower data-range. For example, a signed word less than 8000h saturates to 8000h.
sinusoid	See tone.
Streaming SIMD Extensions	The major enhancement to Intel architecture instruction set. Incorporates a group of general-purpose floating-point instructions operating on packed data, additional packed integer instructions, together with cacheability control and state management instructions. These instructions significantly improve performance of applications using compute-intensive processing of floating-point and integer data.
tone	A sinusoid of a given frequency, phase, and magnitude. Tones are used as test signals and as building blocks for more complex signals.
up-sampling	Up-sampling conceptually increases the signal sampling rate by inserting zero-valued samples between neighboring samples of a signal.
window	A mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

Index

Numerics

10Log10, 5-50

A

AAC

- ADIF header structure, 10-109
- ADTS frame header structure, 10-110
- channel information structure, 10-116
- channel pair element structure, 10-115
- individual channel side information structure, 10-111
- LTP structure, 10-115
- scalable extension element structure, 10-112
- scalable main element structure, 10-111
- TNS structure for one layer, 10-114

about this manual, 1-5

about this software, 1-1

Abs, 5-38

AccCovarianceMatrix, 8-75

accuracy, of arithmetic functions, 12-1

ACELPFixedCodebookSearch_G723, 9-101

Acos, 12-34

Acosh, 12-48

Acoustic Echo Cancellation functions

- CoefUpdateAECNLMS, 8-266
- ControllerGetSizeAEC, 8-269
- ControllerInitAEC, 8-270
- ControllerUpdateAEC, 8-271
- FilterAECNLMS, 8-265
- StepSizeUpdateAECNLMS, 8-268

adaptive codebook, 9-65

adaptive filters, 10-25

AdaptiveCodebookContribution_G729, 9-63

AdaptiveCodebookDecode_AMRWB, 9-205

AdaptiveCodebookDecode_GSMAMR, 9-152

AdaptiveCodebookDecodeGetSize_AMRWB, 9-203

AdaptiveCodebookDecodeInit_AMRWB, 9-203

AdaptiveCodebookDecodeUpdate_AMRWB, 9-204

AdaptiveCodebookGain_G729, 9-64

AdaptiveCodebookGain_GSMAMR, 9-154

AdaptiveCodebookGainCoeff_AMRWB, 9-198

AdaptiveCodebookSearch_AMRWB, 9-200

AdaptiveCodebookSearch_G723, 9-103

AdaptiveCodebookSearch_G729, 9-46

AdaptiveCodebookSearch_GSMAMR, 9-149

Add, 5-17

AddAllRowSum, 8-8

AddC, 5-16

AddMulColumn, 8-175

AddMulRow, 8-176

AddNRows, 8-94

AddProduct, 5-20

AddProductC, 5-21

Adler32, 13-75

ALawToLin, 9-290

ALawToMuLaw, 9-294

AlgebraicCodebookDecode_AMRWB, 9-197

AlgebraicCodebookSearch_AMRWB, 9-192

- AlgebraicCodebookSearch_GSMAMR, 9-156
- AlignPtr(). See ippAlignPtr
- AltInitMCRA, 8-250
- amplitude ratio, 9-109
- AMR WB speech codec
 - AdaptiveCodebookDecode_AMRWB, 9-205
 - AdaptiveCodebookDecodeGetSize_AMRWB, 9-203
 - AdaptiveCodebookDecodeInit_AMRWB, 9-203
 - AdaptiveCodebookDecodeUpdate_AMRWB, 9-204
 - AdaptiveCodebookGainCoeff_AMRWB, 9-198
 - AdaptiveCodebookSearch_AMRWB, 9-200
 - AlgebraicCodebookDecode_AMRWB, 9-197
 - AlgebraicCodebookSearch_AMRWB, 9-192
 - DecDTXBuffer_AMRWB, 9-216
 - DecodeGain_AMRWB, 9-213
 - Deemphasize_AMRWB, 9-188
 - EncDTXBuffer_AMRWB, 9-215
 - GainQuant_AMRWB, 9-211
 - HighPassFilter_AMRWB, 9-186
 - HighPassFilterGetSize_AMRWB, 9-184
 - HighPassFilterInit_AMRWB, 9-185
 - ISFQuant_AMRWB, 9-206
 - ISFQuantDecode_AMRWB, 9-208
 - ISFQuantDecodeDTX_AMRWB, 9-210
 - ISFQuantDTX_AMRWB, 9-209
 - ISFToISP_AMRWB, 9-181
 - ISPToISF_Norm_AMRWB, 9-180
 - ISPToLPC_AMRWB, 9-178
 - LPCToISP_AMRWB, 9-177
 - OpenLoopPitchSearch_AMRWB, 9-182
 - Preemphasize_AMRWB, 9-187
 - ResidualFilter_AMRWB, 9-176
 - SynthesisFilter_AMRWB, 9-189
 - VAD_AMRWB, 9-191
 - VADGetSize_AMRWB, 9-190
 - VADInit_AMRWB, 9-191
- AnalysisFilter_SBR, 10-166
- AnalysisFilterGetSize_SBR, 10-159
- AnalysisFilterInit_SBR, 10-162
- AnalysisPQMF_MP3, 10-67
- And, 5-7
- AndC, 5-6
- Arctan, 5-52
- Arithmetic functions, 5-15 thru 5-49
- 10Log10, 5-50
- Abs, 5-38
- Add, 5-17
- AddC, 5-16
- AddProduct, 5-20
- AddProductC, 5-21
- Arctan, 5-52
- Cubrt, 5-44
- Div, 5-35
- DivC, 5-32
- DivCRev, 5-34
- Exp, 5-45
- Ln, 5-48
- Mul, 5-24
- MulC, 5-22
- Sqr, 5-39
- Sqrt, 5-41
- Sub, 5-30
- SubC, 5-27
- SubCRev, 5-28
- SumLn, 5-51
- Asin, 12-36
- Asinh, 12-50
- Atan, 12-38
- Atan2, 12-40
- Atanh, 12-52
- audience for this manual, 1-7
- audio coding, 10-1
- Audio coding functions
 - CalcSF, 10-13
 - Deinterleave, 10-8
 - FDPFree, 10-31
 - FDPFwd, 10-34
 - FDPGetSize, 10-31
 - FDPInit, 10-30
 - FDPInitAlloc, 10-29
 - FDPInv, 10-35
 - FDPReset, 10-32
 - FDPResetGroup, 10-34
 - FDPResetSfb, 10-33
 - FIRBlockFree, 10-26
 - FIRBlockInitAlloc, 10-25
 - FIRBlockOne, 10-27
 - IndexSelect_VQ, 10-55
 - Interleave, 10-7

- AutoCorr_NormE, 9-17
- AutoCorr_NormE_G723, 9-93
- autocorrelation, in speech codecs, 9-27
- AutoCorrLagMax, 9-15
- AutoScale, 9-12

- bandwidth expansion, 9-96
- Basic arithmetic functions for speech recognition, 8-8 thru 8-17
 - AddAllRowSum, 8-8
 - BlockDMatrixFree, 8-17
 - BlockDMatrixInitAlloc, 8-16
 - CopyColumn_Indirect, 8-14
 - MatVecMul, 8-20
 - SubRow, 8-13
 - SumColumn, 8-9
 - SumRow, 8-11
 - VecMatMul, 8-19
- BhatDist, 8-164
- binomial window, 9-92
- bit stream, 10-13
- BitReservoirInit_MP3, 10-92
- BlindEqualization_Aurora, 8-228
- block filtering, 10-25
- BlockDMatrixFree, 8-17
- BlockDMatrixInitAlloc, 8-16
- BuildSignTable, 8-132
- BuildSymlTableDV4D, 5-101
- Burrows-Wheeler transform, 13-93 thru 13-100
 - BWTFwd, 13-95
 - BWTFwd_SmallBlock, 13-98
 - BWTFwdGetSize, 13-95
 - BWTGetSize_SmallBlock, 13-98
 - BWTInv, 13-97
 - BWTInv_SmallBlock, 13-99
 - BWTInvGetSize, 13-96
- BWT. See Burrows-Wheeler transform
- BWTFwd, 13-95
- BWTFwd_SmallBlock, 13-98
- BWTFwdGetSize, 13-95

BWTGetSize_SmallBlock, 13-98

BWTInv, 13-97

BWTInv_SmallBlock, 13-99

BWTInvGetSize, 13-96

C

CalcSF, 10-13

CalcStatesDV, 5-100

CartToPolar, complex, 5-90

Cbirt, 12-12

CCS format, 7-6 thru ??

CdbkFree, 8-195

CdbkGetSize, 8-190

CdbkInit, 8-191

CdbkInitAlloc, 8-193

CepstrumToLP, 8-33

Chebyshev polynomials, 9-31

codebook, 9-33

fixed, 9-59

indices, 9-37

search, 9-63

CodebookSearch_G728, 9-257

CoefUpdateAECNLMS, 8-266

CombinedFilter_G728, 9-243

CombinedFilterGetStateSize_G728, 9-242

CombinedFilterInit_G728, 9-243

common functions

ippAlignPtr, 3-12

ippFree, 3-15

ippGetCpuClocks, 3-8

ippGetCpuFreqMhz, 3-9

ippGetCpuType, 3-7

ippGetMaxCacheSizeB, 3-9

ippGetNumThreads, 3-13

ippGetStatusString, 3-6

ippMalloc, 3-14

ippSetDenormAreZeros, 3-11

ippSetFlushToZero, 3-10

ippSetNumThreads, 3-13

Companding functions, 9-288 thru 9-295

ALawToLin, 9-290

bALawToMuLaw, 9-294

LinToALaw, 9-291

LinToMuLaw, 9-289

MuLawToALaw, 9-293

MuLawToLin, 9-288

Compare, 11-11

CompareIgnoreCase, 11-13

CompareIgnoreCaseLatin, 11-13

CompensateOffset, 8-23

Concat, 11-22

ConcatC, 11-23

concepts

IPP structures, 2-6

of IPP, 2-1

Conj, 5-63

ConjCcs, 7-11

ConjFlip, 5-64

ConjPack, 7-7

ConjPerm, 7-9

ControllerGetSizeAEC, 8-269

ControllerInitAEC, 8-270

ControllerUpdateAEC, 8-271

Conv, 6-2

ConvCyclic, 6-5

conventions

font, 1-7

naming, 1-8

signal name, 1-8

Conversion Functions

FindNearestOne, 5-97

Conversion functions, 5-54 thru 5-95

CartToPolar, complex, 5-90

Conj, 5-63

ConjFlip, 5-64

Convert, 5-57

CplxToReal, 5-73

FindNearest, 5-98

Flip, 5-96

Imag, 5-71

Join, 5-59

JoinScaled, 5-60

Magnitude, 5-65

MagSquared, 5-67
MaxOrder, 5-94
PolarToCart, complex, 5-92
Preemphasize, 5-95
Real, 5-70
RealToCplx, 5-72
SortAscend, 5-55
SortDescend, 5-55
SplitScaled, 5-61
SwapBytes, 5-56
Threshold, 5-74
Threshold_GT, 5-77
Threshold_GTAbs, 5-80
Threshold_GTVal, 5-84
Threshold_LT, 5-77
Threshold_LTAbs, 5-80
Threshold_LTInv, 5-88
Threshold_LTVal, 5-84
Threshold_LTValGTVal, 5-84
Convert, 5-57
Convolution and correlation functions, 6-1 thru 6-10
 Conv, 6-2
 ConvCyclic, 6-5
 CrossCorr, 6-8
 UpdateLinear, 6-10
 UpdatePower, 6-11
ConvPartial, 9-6
Copy, 4-3
CopyColumn, 8-77
CopyColumn_Indirect, 8-14
CopyWithPadding, 8-47
Cos, 12-27
Cosh, 12-43
CplxToReal, 5-73
CRC32, 13-76
Cross-Architecture Alignment, 1-2
CrossCorr, 6-8, 9-18
CrossCorrCoeff, 8-90
CrossCorrCoeffDecim, 8-89
CrossCorrCoeffInterpolation, 8-92
CrossCorrLagMax, 9-19
cross-platform applications, 2-2
Cubrt, 5-44

D

Data compression functions

Adler32, 13-75
BWTFwd, 13-95
BWTFwd_SmallBlock, 13-98
BWTFwdGetSize, 13-95
BWTGetSize_SmallBlock, 13-98
BWTInv, 13-97
BWTInv_SmallBlock, 13-99
BWTInvGetSize, 13-96
CRC32, 13-76
DecodeGIT, 13-111
DecodeGITGetSize, 13-111
DecodeGITInit, 13-110
DecodeGITInitAlloc, 13-109
DecodeHuff, 13-27
DecodeHuffInit, 13-26
DecodeHuffInitAlloc, 13-25
DecodeHuffOne, 13-27
DecodeLZ77, 13-66
DecodeLZ77DynamicHuff, 13-69
DecodeLZ77FixedHuff, 13-68
DecodeLZ77GetPairs, 13-72
DecodeLZ77GetSize, 13-64
DecodeLZ77GetStatus, 13-73
DecodeLZ77Init, 13-64
DecodeLZ77InitAlloc, 13-65
DecodeLZ77InitGetBlockType, 13-67
DecodeLZ77Reset, 13-75
DecodeLZ77SelectHuffMode, 13-57
DecodeLZ77SetPairs, 13-73
DecodeLZ77SetStatus, 13-74
DecodeLZ77StoredBlock, 13-71
DecodeLZSS, 13-42
DecodeLZSSInit, 13-41
DecodeLZSSInitAlloc, 13-40
DecodeRLE, 13-121
EncodeGIT, 13-105
EncodeGITGetSize, 13-104
EncodeGITInit, 13-103
EncodeGITInitAlloc, 13-101
EncodeHuff, 13-23
EncodeHuffFinal, 13-24
EncodeHuffInit, 13-20
EncodeHuffInitAlloc, 13-19

EncodeHuffOne, 13-22
EncodeLZ77, 13-53
EncodeLZ77FixedHuff, 13-55
EncodeLZ77Flush, 13-59
EncodeLZ77GetPairs, 13-60
EncodeLZ77GetSize, 13-51
EncodeLZ77GetStatus, 13-61
EncodeLZ77Init, 13-50
EncodeLZ77InitAlloc, 13-52
EncodeLZ77Reset, 13-63
EncodeLZ77SelectHuffMode, 13-54
EncodeLZ77SetPairs, 13-61
EncodeLZ77SetStatus, 13-62
EncodeLZ77StoredBlock, 13-58
EncodeLZSS, 13-36
EncodeLZSSFlush, 13-37
EncodeLZSSInit, 13-35
EncodeLZSSInitAlloc, 13-34
EncodeRLE, 13-120
GITFree, 13-102
HuffFree, 13-20
HuffGetDstBufSize, 13-28
HuffGetLenCodeTable, 13-24
HuffGetSize, 13-21
HuffLenCodeTablePack, 13-29
HuffLenCodeTableUnpack, 13-30
LZ77Free, 13-53
LZSSFree, 13-34
LZSSGetSize, 13-36
MTFFwd, 13-118
MTFGetSize, 13-118
MTFInit, 13-117
MTFInitAlloc, 13-116, 13-117
MTFInv, 13-119
VLCCountBits, 13-10
VLCDecodeBlock, 13-15
VLCDecodeFree, 13-12
VLCDecodeGetSize, 13-14
VLCDecodeInit, 13-12
VLCDecodeInitAlloc, 13-11
VLCDecodeOne, 13-16
VLCEncodeBlock, 13-8
VLCEncodeFree, 13-6
VLCEncodeGetSize, 13-7
VLCEncodeInit, 13-6
VLCEncodeInitAlloc, 13-5
VLCEncodeOne, 13-9
DesClustLAccumulate, 8-172
DesClustLCompute, 8-173
DCT functions, 7-63 thru 7-69
 DCTFwd, 7-67
 DCTFwdFree, 7-65
 DCTFwdGetBufSize, 7-66
 DCTFwdInitAlloc, 7-63
 DCTInv, 7-67
 DCTInvFree, 7-65
 DCTInvGetBufSize, 7-66
 DCTInvInitAlloc, 7-63
DCTFwd, 7-67
DCTFwd_G722, 9-226
DCTFwdFree, 7-65
DCTFwdGetBufSize, 7-66
DCTFwdInitAlloc, 7-63
DCTInv, 7-67
DCTInv_G722, 9-226
DCTInvFree, 7-65
DCTInvGetBufSize, 7-66
DCTInvInitAlloc, 7-63
DCTLifter, 8-66
DCTLifterFree, 8-66
DCTLifterGetSize_MulC0, 8-62
DCTLifterInit_MulC0, 8-63
DCTLifterInitAlloc, 8-64
DecDTXBuffer_AMRWB, 9-216
DecDTXBuffer_GSMAMR, 9-170
Decode_G726, 9-235
DecodeAdaptiveVector_G723, 9-117
DecodeAdaptiveVector_G729, 9-49
DecodeChanPairElt_AAC, 10-122
DecodeDatStrElt_AAC, 10-127
DecodeExtensionHeader_AAC, 10-143
DecodeFillElt_AAC, 10-128
DecodeGain_AMRWB, 9-213
DecodeGain_G729, 9-58
DecodeGetStateSize_G726, 9-234
DecodeGIT, 13-111

- DecodeGITGetSize, 13-111
- DecodeGITInit, 13-110
- DecodeGITInitAlloc, 13-109
- DecodeHuffInit, 13-26
- DecodeHuffInitAlloc, 13-25
- DecodeHuffOne, 13-27
- DecodeInit_G726, 9-234
- DecodeIsStereo_AAC, 10-133
- DecodeLZ77, 13-66
- DecodeLZ77DynamicHuff, 13-69
- DecodeLZ77FixedHuff, 13-68
- DecodeLZ77GetBlockType, 13-67
- DecodeLZ77GetPairs, 13-72
- DecodeLZ77GetSize, 13-64
- DecodeLZ77GetStatus, 13-73
- DecodeLZ77Init, 13-64
- DecodeLZ77InitAlloc, 13-65
- DecodeLZ77Reset, 13-75
- DecodeLZ77SetStatus, 13-74
- DecodeLZ77StoredBlock, 13-71
- DecodeLZSS, 13-42
- DecodeLZSSInit, 13-41
- DecodeLZSSInitAlloc, 13-40
- DecodeMainHeader_AAC, 10-142
- DecodeMsPNS_AAC, 10-146
- DecodeMsStereo_AAC, 10-131
- DecodePNS_AAC, 10-145
- DecodePrgCfgElt_AAC, 10-121
- DecodeRLE, 13-121
- DecodeTNS_AAC, 10-136
- DecomposeDCTToMLT, 9-229
- DecomposeMLTToDCT, 9-227
- Deemphasize_AMRWB, 9-188
- Deemphasize_GSMFR, 9-219
- DEFLATE, 13-45
- Deinterleave, 10-8
- DeinterleaveSpectrum_AAC, 10-135
- Delta, 8-80
- DeltaDelta, 8-85
- Derivative functions, 8-77 thru 8-89
 - AccCovarianceMatrix, 8-75
 - CopyColumn, 8-77
 - Delta, 8-80
 - DeltaDelta, 8-85
 - EvalDelta, 8-78
- DFT for given frequency (Goertzel) functions, 7-59 thru 7-63
 - Goertz, 7-60
 - GoertzTwo, 7-62
- DFT functions, 7-40 thru 7-59
 - DFTFree_C, 7-42
 - DFTFree_R, 7-42
 - DFTFwd_CToC, 7-45
 - DFTFwd_RToCCS, 7-49
 - DFTFwd_RToPack, 7-49
 - DFTFwd_RToPerm, 7-49
 - DFTGetBufSize_C, 7-43
 - DFTGetBufSize_R, 7-43
 - DFTInitAlloc_C, 7-40
 - DFTInitAlloc_R, 7-40
 - DFTInv_CCSToR, 7-52
 - DFTInv_CToC, 7-47
 - DFTInv_PackToR, 7-52
 - DFTInv_PermToR, 7-52
 - DFTOutOrdFree_C, 7-55
 - DFTOutOrdFwd_CToC, 7-57
 - DFTOutOrdGetBufSize_C, 7-56
 - DFTOutOrdInitAlloc_C, 7-54
 - DFTOutOrdInv_CToC, 7-58
- DFTInv_CToC, 7-47
- dispatcher functions
 - ippStaticInit, 3-16
 - ippStaticInitCpu, 3-17
- Distance coding, 13-100
- Div, 5-32, 5-35, 12-6
- DivCRev, 5-34
- DotProd, 5-136
- DotProd_G729, 9-24
- DotProdAutoScale, 9-13
- DotProdColumn, 8-178
- DTW, 8-139
- Durbin, 8-28

E

EMNS functions

- AltInitMCRA, 8-250
- FilterUpdateEMNS, 8-243
- FilterUpdateWiener, 8-246
- GetSizeMCRA, 8-248
- InitMCRA, 8-249
- UpdateNoisePSDMCRA, 8-251

EmptyFBankInitAlloc, 8-56

EncDTXBuffer_AMRWB, 9-215

EncDTXBuffer_GSMAMR, 9-170

EncDTXHandler_GSMAMR, 9-168

EncDTXSID_GSMAMR, 9-166

Encode_G726, 9-233

EncodeGetStateSize_G726, 9-231

EncodeGIT, 13-105

EncodeGITGetSize, 13-104

EncodeGITInit, 13-103

EncodeGITInitAlloc, 13-101

EncodeHuff, 13-23

EncodeHuffFinal, 13-24

EncodeHuffInit, 13-20

EncodeHuffInitAlloc, 13-19

EncodeHuffOne, 13-22

EncodeInit_G726, 9-232

EncodeLZ77, 13-53

EncodeLZ77DynamicHuff, 13-57

EncodeLZ77FixedHuff, 13-55

EncodeLZ77Flush, 13-59

EncodeLZ77GetPairs, 13-60, 13-73

EncodeLZ77GetSize, 13-51

EncodeLZ77GetStatus, 13-61

EncodeLZ77Init, 13-50

EncodeLZ77InitAlloc, 13-52

EncodeLZ77Reset, 13-63

EncodeLZ77SelectHuffMode, 13-54

EncodeLZ77SetPairs, 13-61

EncodeLZ77SetStatus, 13-62

EncodeLZ77StoredBlock, 13-58

EncodeLZSS, 13-36

EncodeLZSSFlush, 13-37

EncodeLZSSInit, 13-35

EncodeLZSSInitAlloc, 13-34

EncodeRLE, 13-120

EncodeTNS_AAC, 10-151

energy, average, 9-62

Entropy, 8-161

Equal, 11-14

Erf, 12-54

Erfc, 12-56

error reporting, 2-12

EvalDelta, 8-78

EvalDelta_Aurora, 8-229

EvalFBank, 8-61

excitation, 9-48

- random, 9-89

Exp, 5-45, 12-21

ExpNegSqr, 8-163

F

FBankFree, 8-57

FBankGetCenters, 8-57

FBankGetCoeffs, 8-59

FBankSetCenters, 8-58

FBankSetCoeffs, 8-60

FDPFree, 10-31

FDPFwd, 10-34

FDPGetSize, 10-31

FDPInit, 10-30

FDPInitAlloc, 10-29

FDPInv, 10-35

FDPReset, 10-32

FDPResetGroup, 10-34

FDPResetSfb, 10-33

Feature processing functions, 8-22 thru 8-93

CepstrumToLP, 8-33

CompensateOffset, 8-23

CopyWithPadding, 8-47

DCTLifter, 8-66

- DCTLifterFree, 8-66
- DCTLifterGetSize_MulC0, 8-62
- DCTLifterInit_MulC0, 8-63
- DCTLifterInitAlloc, 8-64
- Durbin, 8-28
- EmptyFBankInitAlloc, 8-56
- EvalFBank, 8-61
- FBankFree, 8-57
- FBankGetCenters, 8-57
- FBankGetCoeffs, 8-59
- FBankSetCenters, 8-58
- FBankSetCoeffs, 8-60
- LinearPrediction, 8-26
- LinearToMel, 8-46
- LPToCepstrum, 8-32
- LPToLSP, 8-41
- LPToReflection, 8-34
- LPToSpectrum, 8-31
- MelFBankGetSize, 8-48
- MelFBankInit, 8-49
- MelFBankInitAlloc, 8-50
- MelLinFBankInitAlloc, 8-53
- MelToLinear, 8-45
- PitchmarkToF0, 8-39
- ReflectionToAR, 8-36
- ReflectionToLP, 8-35
- ReflectionToTilt, 8-38
- Schur, 8-29
- SignChangeRate, 8-25
- UnitCurve, 8-40
- ZeroMean, 8-22
- FFT functions, 7-18 thru 7-39
 - FFTFree_C, 7-20
 - FFTFree_R, 7-20
 - FFTFwd_CToC, 7-27
 - FFTFwd_RToCCS, 7-33
 - FFTFwd_RToPack, 7-33
 - FFTFwd_RToPerm, 7-33
 - FFTGetBufSize_C, 7-26
 - FFTGetBufSize_R, 7-26
 - FFTGetSize_C, 7-24
 - FFTGetSize_R, 7-24
 - FFTInit_C, 7-22
 - FFTInit_R, 7-22
 - FFTInitAlloc_C, 7-18
 - FFTInitAlloc_R, 7-18
 - FFTInv_CCSToR, 7-37
 - FFTInv_CToC, 7-30
 - FFTInv_PackToR, 7-37
 - FFTInv_PermToR, 7-37
- FFTFwd_RToPerm_GSMAMR, 9-124
- FillShortlist_Column, 8-136
- FillShortlist_Row, 8-134
- filter
 - inverse, 9-74
 - long-term, 9-71
 - short-term, 9-74
- FilterAECNLMS, 8-265
- FilteredExcitation_G729, 9-89
- Filtering functions, 6-13 thru 6-127
 - SumWindow, 6-16
- FilterMedian, 6-126
- FilterUpdateEMNS, 8-243
- FilterUpdateWiener, 8-246
- Find, FindRev, 11-3
- FindC, FindRevC, 11-5
- FindCAny, FindRevCAny, 11-7
- FindNearest, 5-98
- FindNearestOne, 5-97
- FindPeaks, 8-274
- FIR, 6-40
- FIR filter functions, 6-17 thru 6-66
 - FIR, 6-40
 - FIR_Direct, 6-48
 - FIRFree, 6-23
 - FIRGenBandpass, 6-63
 - FIRGenBandstop, 6-65
 - FIRGenHighpass, 6-62
 - FIRGenLowpass, 6-61
 - FIRGetDlyLine, 6-35
 - FIRGetStateSize, 6-29
 - FIRGetTaps, 6-32
 - FIRInit, 6-24
 - FIRInitAlloc, 6-18
 - FIRMR_Direct, 6-52
 - FIRMRGetStateSize, 6-29
 - FIRMRInit, 6-26
 - FIRMRInitAlloc, 6-20
 - FIROne, 6-38

- FIROne_Direct, 6-45
- FIRSetDlyLine, 6-37
- FIRSetTaps, 6-33
- FIRSparse, 6-59
- FIRSparseGetStateSize, 6-58
- FIRSparseInit, 6-56
- FIR LMS filter functions, 6-66 thru 6-89
- FIR_Direct, 6-48
- FIRBlockFree, 10-26
- FIRBlockInitAlloc, 10-25
- FIRBlockOne, 10-27
- FIRFree, 6-23
- FIRGenBandpass, 6-63
- FIRGenBandstop, 6-65
- FIRGenHighpass, 6-62
- FIRGenLowpass, 6-61
- FIRGetDlyLine, 6-35
- FIRGetStateSize, 6-29
- FIRGetTaps, 6-32
- FIRInit, 6-24
- FIRInitAlloc, 6-18
- FIRLMS, 6-71
- FIRLMSFree, 6-68
- FIRLMSGGetDlyLine, 6-69
- FIRLMSGGetTaps, 6-68
- FIRLMSInitAlloc, 6-67
- FIRLMSMRFree, 6-78
- FIRLMSMRGetDlyVal, 6-86
- FIRLMSMRGetTapsPointer, 6-83
- FIRLMSMRInitAlloc, 6-77
- FIRLMSMROne, 6-88
- FIRLMSMROneVal, 6-89
- FIRLMSMRPutVal, 6-87
- FIRLMSMRSetMu, 6-79
- FIRLMSMRUpdateTaps, 6-80
- FIRLMSOne_Direct, 6-74
- FIRLMSSetDlyLine, 6-70
- FIRMR_Direct, 6-52
- FIRMRGetStateSize, 6-29
- FIRMRInitAlloc, 6-20
- FIRMTInit, 6-26
- FIROne, 6-38
- FIROne_Direct, 6-45
- FIRSetDlyLine, 6-37
- FIRSetTaps, 6-33
- FIRSparse, 6-59
- FIRSparseGetStateSize, 6-58
- FIRSparseInit, 6-56
- Fixed filter banks wavelet transforms, 7-77 thru 7-81
- fixed-accuracy arithmetic functions
 - exponential and logarithmic
 - Exp, 12-21
 - Ln, 12-23
 - Log10, 12-25
 - hyperbolic
 - Acosh, 12-48
 - Asinh, 12-50
 - Atanh, 12-52
 - Cosh, 12-43
 - Sinh, 12-45
 - Tanh, 12-47
 - power and root
 - Cbrt, 12-12
 - Div, 12-6
 - Inv, 12-4
 - InvCbrt, 12-13
 - InvSqrt, 12-10
 - Pow, 12-15
 - Powx, 12-18
 - Sqrt, 12-8
 - special
 - Erf, 12-54
 - Erfc, 12-56
 - trigonometric
 - Acos, 12-34
 - Asin, 12-36
 - Atan, 12-38
 - Atan2, 12-40
 - Cos, 12-27
 - Sin, 12-29
 - SinCos, 12-30
 - Tan, 12-32
- FixedCodebookDecode_GSMAMR, 9-159
- FixedCodebookSearch_G729, 9-50

Flag and hint arguments, 7-4

Flip, 5-96

font conventions, 1-7

FormVector, 8-187

FormVectorVQ, 8-201

fraction delay, 9-47

frequency domain prediction, 10-29

function descriptions, 1-6

function naming, 2-2

G

G.722.1 speech codec

DCTFwd_G722, DCTInv_G722, 9-226

DecomposeDCTToMLT, 9-229

DecomposeMLTToDCT, 9-227

HuffmanEncode_G722, 9-230

G.723.1 speech codec

codebook search functions

ACELPFixedCodebookSearch_G723, 9-101

AdaptiveCodebookSearch_G723, 9-103

MPMLQFixedCodebookSearch_G723, 9-104

OpenLoopPitchSearch_G723, 9-100

ToeplitzMatrix_G723, 9-106

filter functions

DecodeAdaptiveVector_G723, 9-117

HarmonicNoiseSubtract_G723, 9-116

HarmonicSearch_G723, 9-114

HighPassFilter_G723, 9-110

IIR16s_G723, 9-111

PitchPostFilter_G723, 9-118

SynthesisFilter_G723, 9-112

TiltCompensation_G723, 9-114

gain quantization functions

GainControl_G723, 9-109

GainQuant_G723, 9-107

LP analysis functions

AutoCorr_G723, 9-92

AutoCorr_NormE_G723, 9-93

LevinsonDurbin_G723, 9-94

LPCToLSF_G723, 9-96

LSFDecode_G723, 9-98

LSFQuant_G723, 9-99

LSFToLPC_G723, 9-97

G.726 speech codec

Decode_G726, 9-235

DecodeGetStateSize_G726, 9-234

DecodeInit_G726, 9-234

Encode_G726, 9-233

EncodeGetStateSize_G726, 9-231

EncodeInit_G726, 9-232

G.728 speech codec

CodebookSearch_G728, 9-257

CombinedFilter_G728, 9-243

CombinedFilterGetStateSize_G728, 9-242

CombinedFilterInit_G728, 9-243

IIR_G728, 9-239

IIR_Init_G728, 9-238

IIRGetStateSize_G728, 9-238

ImpulseResponseEnergy_G728, 9-258

LevinsonDurbin_G728, 9-255

LPCInverseFilter_G728, 9-249

PitchPeriodExtraction_G728, 9-250

PostFilter_G728, 9-246

PostFilterAdapterGetStateSize_G728, 9-247

PostFilterAdapterStateInit_G728, 9-248

PostFilterGetStateSize_G728, 9-245

PostFilterInit_G728, 9-245

SynthesisFilter_G728, 9-241

SynthesisFilterGetStateSize_G728, 9-240

SynthesisFilterInit_G728, 9-240

WinHybrid_G728, 9-254

WinHybridGetStateSize_G728, 9-252

WinHybridInit_G728, 9-253

G.729 codec functions, 9-22

G.729 speech codec

basic functions

DotProd_G729, 9-24

Interpolate_G729, 9-25

codebook search functions

AdaptiveCodebookContribution_G729, 9-63

AdaptiveCodebookGain_G729, 9-64

AdaptiveCodebookSearch_G729, 9-46

DecodeAdaptiveVector_G729, 9-49

FixedCodebookSearch_G729, 9-50

LagWindow_G729, 9-41

OpenLoopPitchSearch_G729, 9-43

ToeplitzMatrix_G729, 9-55

filter functions

-
- FilteredExcitation_G729, 9-89
 - HarmonicFilter, 9-77
 - HighPassFilter_G729, 9-80
 - HighPassFilterInit_G729, 9-79
 - HighPassFilterSize_G729, 9-78
 - IIR16s_G729, 9-81
 - LongTermPostFilter_G729, 9-70
 - PhaseDispersion_G729D, 9-84
 - PhaseDispersionGetStateSize_G729D, 9-82
 - PhaseDispersionInit_G729D, 9-82
 - PhaseDispersionUpdate_G729D, 9-83
 - Preemphasize_G729, 9-85
 - RandomNoiseExcitation_G729B, 9-88
 - ResidualFilter_G729, 9-67
 - ShortTermPostFilter_G729, 9-73
 - SynthesisFilter_G729, 9-68
 - TiltCompensation_G729, 9-75
 - WinHybrid_G729E, 9-87
 - WinHybridGetStateSize_G729E, 9-86
 - WinHybridInit_G729E, 9-86
 - gain quantization functions
 - DecodeGain_G729, 9-58
 - GainControl_G729, 9-59
 - GainQuant_G729, 9-61
 - LP analysis functions
 - AutoCorr_G729, 9-27
 - LevinsonDurbin_G729, 9-28
 - LPCToLSP_G729, 9-30
 - LSFDecode_G729, 9-34
 - LSFDecodeErased_G729, 9-35
 - LSFQuant_G729, 9-32
 - LSFToLSP_G729, 9-32
 - LSPQuant_G729, 9-37
 - LSPToLPC_G729, 9-36
 - LSPToLSF_G729, 9-40
 - gain
 - estimation, 9-108
 - of the adaptive-codebook vector, 9-65
 - optimum, 9-62
 - GainControl_G723, 9-109
 - GainControl_G729, 9-59
 - GainQuant_AMRWB, 9-211
 - GainQuant_G723, 9-107
 - GainQuant_G729, 9-61
 - Gaussian distribution functions
 - RandGauss, 4-35
 - RandGauss_Direct, 4-36
 - RandGaussFree, 4-32
 - RandGaussGetSize, 4-33
 - RandGaussInit, 4-34
 - RandGaussInitAlloc, 4-31
 - GaussianDist, 8-157
 - GaussianMerge, 8-160
 - GaussianSplit, 8-159
 - generalized interval transformation (GIT) coding, 13-100
 - thru ??
 - Generalized Interval Transformation coding, 13-100
 - GetCdbkSize, 8-195
 - GetCodebook, 8-196
 - GetCpuClocks(). See `ippGetCpuClocks`
 - GetCpuFreqMhz(). See `ippGetCpuFreqMhz`
 - GetCpuType(). See `ippGetCpuType`
 - GetLibVersion, 3-2
 - GetMaxCacheSizeB(). See `ippGetMaxCacheSizeB`
 - GetNumThreads(). See `ippGetNumThreads`
 - GetSizeMCRA, 8-248
 - GetStatusString(). See `ippGetStatusString`
 - GetVarPointDV, 5-99
 - GIT coding. See `generalized interval transformation coding`
 - GITFree, 13-102
 - Goertz, 7-60
 - GoertzTwo, 7-62
 - GSM FR speech codec
 - Deemphasize_GSMFR, 9-219
 - HighPassFilter_GSMFR, 9-222
 - Preemphasize_GSMFR, 9-225
 - RPEQuantDecode_GSMFR, 9-218
 - Schur_GSMFR, 9-223
 - ShortTermAnalysisFilter_GSMFR, 9-220
 - ShortTermSynthesisFilter_GSMFR, 9-221
 - WeightingFilter_GSMFR, 9-224
 - GSM-AMR speech codec, 9-121
 - adaptive codebook primitives, 9-136
 - AdaptiveCodebookDecode_GSMAMR, 9-152
 - AdaptiveCodebookGain_GSMAMR, 9-154
 - AdaptiveCodebookSearch_GSMAMR, 9-149

AlgebraicCodebookSearch_GSMAMR, 9-156
AutoCorr_GSMAMR, 9-125
data structures, 9-4
DecDTXBuffer_GSMAMR, 9-170
discontinuous transmission (DTX) primitives, 9-160
EncDTXBuffer_GSMAMR, 9-170
EncDTXHandler_GSMAMR, 9-168
EncDTXSID_GSMAMR, 9-166
FFTFwd_RToPerm_GSMAMR, 9-124
fixed codebook primitives, 9-155
FixedCodebookDecode_GSMAMR, 9-159
ImpulseResponseTarget_GSMAMR, 9-147
Interpolate_GSMAMR, 9-123
LevinsonDurbin_GSMAMR, 9-127
LP analysis and quantization primitives, 9-125
LPCToLSP_GSMAMR, 9-129
LSFToLSP_GSMAMR, 9-132
LSPQuant_GSMAMR, 9-132
LSPToLPC_GSMAMR, 9-130
open-loop pitch search, 9-137
OpenLoopPitchSearchDTXVAD1_GSMAMR,
9-141
OpenLoopPitchSearchDTXVAD2_GSMAMR,
9-144
OpenLoopPitchSearchNonDTX_GSMAMR, 9-139
PostFilter_GSMAMR, 9-171
Preemphasize_GSMAMR, 9-160
QuantLSPDecode_GSMAMR, 9-135
VAD1_GSMAMR, 9-161
VAD2_GSMAMR, 9-164
GZIP, 13-45

H

Hamming window, 9-92
hardware and software requirements, 1-2
HarmonicFilter, 9-77
HarmonicNoiseSubtract_G723, 9-116
HarmonicSearch_G723, 9-114
Hash, 11-21
HighBandCoding_Aurora, 8-226
HighPassFilter_AMRWB, 9-186
HighPassFilter_G723, 9-110
HighPassFilter_G729, 9-80

HighPassFilter_GSMFR, 9-222
HighPassFilterGetSize_AMRWB, 9-184
HighPassFilterInit_AMRWB, 9-185
HighPassFilterInit_G729, 9-79
HighPassFilterSize_G729, 9-78
Hilbert, 7-72
Hilbert transform functions, 7-70 thru 7-73
Hilbert, 7-72
HilbertFree, 7-71
HilbertInitAlloc, 7-70
HilbertFree, 7-71
HilbertInitAlloc, 7-70
HuffFree, 13-20
HuffGetLenCodeTable, 13-24
HuffGetSize, 13-21
HuffLenCodeTablePack, 13-29
HuffLenCodeTableUnpack, 13-30
Huffman coding, ?? thru 13-31
DecodeHuff, 13-27
DecodeHuffInit, 13-26
DecodeHuffInitAlloc, 13-25
DecodeHuffOne, 13-27
EncodeHuff, 13-23
EncodeHuffFinal, 13-24
EncodeHuffInit, 13-20
EncodeHuffInitAlloc, 13-19
EncodeHuffOne, 13-22
HuffFree, 13-20
HuffGetDstBuffSize, 13-28
HuffGetLenCodeTable, 13-24
HuffGetSize, 13-21
HuffLenCodeTablePack, 13-29
HuffLenCodeTableUnpack, 13-30
HuffmanDecode_MP3, 10-99
HuffmanEncode_G722, 9-230
HuffmanEncode_MP3, 10-86

I

IIR, 6-111
IIR filter functions, 6-90 thru 6-124
IIR, 6-111
IIR_BiQuadDirect, 6-119

- IIR_Direct, 6-118
- IIRFree, 6-96
- IIRGetDlyLine, 6-106
- IIRGetStateSize, 6-102
- IIRGetStateSize_BiQuad, 6-103
- IIRInit, 6-97
- IIRInit_BiQuad, 6-100
- IIRInitAlloc, 6-92
- IIRInitAlloc_BiQuad, 6-94
- IIROne, 6-109
- IIROne_BiQuadDirect, 6-117
- IIROne_Direct, 6-116
- IIRSetDlyLine, 6-108
- IIRSetTaps, 6-105
- IIRSparse, 6-123
- IIRSparseGetStateSize, 6-122
- IIRSparseInit, 6-121
- IIR_BiQuadDirect, 6-119
- IIR_Direct, 6-118
- IIR_G728, 9-239
- IIR_Init_G728, 9-238
- IIR16s_G723, 9-111
- IIR16s_G729, 9-81
- IIRFree, 6-96
- IIRGetDlyLine, 6-106
- IIRGetStateSize_BiQuad, 6-103
- IIRGetStateSize_G728, 9-238
- IIRInit, 6-97
- IIRInit_BiQuad, 6-100
- IIRInitAlloc, 6-92
- IIRInitAlloc_BiQuad, 6-94
- IIROne, 6-109
- IIROne_BiQuadDirect, 6-117
- IIROne_Direct, 6-116
- IIRSetDlyLine, 6-108
- IIRSetTaps, 6-105
- IIRSparse, 6-123
- IIRSparseGetStateSize, 6-122
- IIRSparseInit, 6-121
- Imag, 5-71
- ImpulseResponseEnergy_G728, 9-258
- ImpulseResponseTarget_GSMAMR, 9-147
- IndexSelect_VQ, 10-55
- InitMCRA, 8-249
- Insert, 11-8
- Intel Performance Library Suite, 2-1
- Intel Performance Primitives software, 1-1
- Interleave, 10-7
- interleaved to multi-row format conversion, 10-7
- Interpolate_G729, 9-25
- Interpolate_GSMAMR, 9-123
- InterpolateC_NR, 9-7
- Inv, 12-4
- InvCbrt, 12-13
- Inversion frequencies, 13-100
- InvSqrt, 9-14, 12-10
- ippAlignPtr, 3-12
- ippCoreGetCpuType, 3-7
- ippFree, 3-15
- ippGetCpuClocks, 3-8
- ippGetCpuFreqMhz, 3-9
- ippGetGetMaxCacheSizeB, 3-9
- ippGetNumThreads, 3-13
- ippGetStatusString, 3-6
- ippMalloc, 3-14
- ipps10Log10(). See 10Log10
- ippsAbs(). See Abs
- ippsAccCovarianceMatrix(). See AccCovarianceMatrix
- ippsACELPFixedCodebookSearch_G723(). See ACELPFixedCodebookSearch_G723
- ippsAcos, 12-34
- ippsAcosh, 12-48
- ippsAdaptiveCodebookContribution_G729(). See AdaptiveCodebookContribution_G729
- ippsAdaptiveCodebookGain_G729(). See AdaptiveCodebookGain_G729
- ippsAdaptiveCodebookSearch_G723(). See AdaptiveCodebookSearch_G723
- ippsAdaptiveCodebookSearch_G729(). See AdaptiveCodebookSearch_G729
- ippsAdd(). See Add

`ippsAddAllRowSum()`. See `AddAllRowSum`
`ippsAddC()`. See `AddC`
`ippsAddMulColumn()`. See `AddMulColumn`
`ippsAddMulRow()`. See `AddMulRow`
`ippsAddNRows()`. See `AddNRows`
`ippsAddProduct()`. See `AddProduct`
`ippsAddProductC()`. See `AddProductC`
`ippsAdler32()`. See `Adler32`
`ippsALawToLin()`. See `ALawToLin`
`ippsALawToMuLaw()`. See `ALawToMuLaw`
`ippsAltInitMCRA()`. See `AltInitMCRA`
`ippsAnalysisFilter_SBR()`. See `AnalysisFilter_SBR`
`ippsAnalysisFilterGetSize_SBR()`. See `AnalysisFilterGetSize_SBR`
`ippsAnalysisFilterInit_SBR()`. See `AnalysisFilterInit_SBR`
`ippsAnalysisPQMF_MP3()`. See `AnalysisPQMF_MP3`
`ippsAnalysisPQMF_MP3_16s32s`, 10-67
`ippsAnd()`. See `And`
`ippsAndC()`. See `AndC`
`ippsArctan()`. See `Arctan`
`ippsAsin`, 12-36
`ippsAsinh`, 12-50
`ippsAtan`, 12-38
`ippsAtan2`, 12-40
`ippsAtanh`, 12-52
`ippsAutoCorr()`. See `AutoCorr`
`ippsAutoCorr_G723()`. See `AutoCorr_G723`
`ippsAutoCorr_G729()`. See `AutoCorr_G729`
`ippsAutoCorr_NormE()`. See `AutoCorr_NormE`
`ippsAutoCorr_NormE_G723()`. See `AutoCorr_NormE_G723`
`ippsAutoCorrLagMax()`. See `AutoCorrLagMax`
`ippsAutoScale()`. See `AutoScale`
`ippsBhatDist()`. See `BhatDist`
`ippsBitReservoirInit_MP3()`. See `BitReservoirInit_MP3`
`ippsBlindEqualization_Aurora()`. See `BlindEqualization_Aurora`
`ippsBlockDMatrixFree()`. See `BlockDMatrixFree`
`ippsBlockDMatrixInitAlloc()`. See `BlockDMatrixInitAlloc`
`ippsBuildSignTable()`. See `BuildSignTable`
`ippsBuildSymblTableDV4D()`. See `BuildSymblTableDV4D`
`ippsBWTFwd()`. See `BWTFwd`
`ippsBWTFwd_SmallBlock()`. See `BWTFwd_SmallBlock`
`ippsBWTFwdGetSize()`. See `BWTFwdGetSize`
`ippsBWTGetSize_SmallBlock()`. See `BWTGetSize_SmallBlock`
`ippsBWTInv()`. See `BWTInv`
`ippsBWTInv_SmallBlock()`. See `BWTInv_SmallBlock`
`ippsBWTInvGetSize()`. See `BWTInvGetSize`
`ippsCalcSF()`. See `CalcSF`
`ippsCalcStatesDV()`. See `CalcStatesDV`
`ippsCartToPolar()`. See `CartToPolar`, complex
`ippsCbrt`, 12-12
`ippsCdbkFree()`. See `CdbkFree`
`ippsCdbkGetSize()`. See `CdbkGetSize`
`ippsCdbkInit()`. See `CdbkInit`
`ippsCdbkInitAlloc()`. See `CdbkInitAlloc`
`ippsCepstrumToLP()`. See `CepstrumToLP`
`ippsCoefUpdateAECNLMS()`. See `CoefUpdateAECNLMS`
`ippsCompare()`. See `Compare`
`ippsCompareIgnoreCase()`. See `CompareIgnoreCase`
`ippsCompensateOffset()`. See `CompensateOffset`
`ippsConcat()`. See `Concat`
`ippsConcatC()`. See `ConcatC`
`ippsConj()`. See `Conj`
`ippsConjCcs()`. See `ConjCcs`
`ippsConjFlip()`. See `ConjFlip`
`ippsConjPack()`. See `ConjPack`
`ippsControllerGetSizeAEC()`. See `ControllerGetSizeAEC`
`ippsControllerInitAEC()`. See `ControllerInitAEC`
`ippsControllerUpdateAEC()`. See `ControllerUpdateAEC`
`ippsConv()`. See `Conv`
`ippsConvCyclic()`. See `ConvCyclic`
`ippsConvert()`. See `Convert`
`ippsConvPartial()`. See `ConvPartial`

`ippsCopy()`. See `Copy`
`ippsCopyColumn()`. See `CopyColumn`
`ippsCopyColumn_Indirect()`. See `CopyColumn_Indirect`
`ippsCopyWithPadding()`. See `CopyWithPadding`
`ippsCos`, 12-27
`ippsCosh`, 12-43
`ippsCplxToReal()`. See `CplxToReal`
`ippsCRC32()`. See `CRC32`
`ippsCrossCorr()`. See `CrossCorr`
`ippsCrossCorrCoeff()`. See `CrossCorrCoeff`
`ippsCrossCorrCoeffDecim()`. See `CrossCorrCoeffDecim`
`ippsCrossCorrCoeffInterpolation()`. See `CrossCorrCoeffInterpolation`
`ippsCubrt()`. See `Cubrt`
`ippsDcsClustLAccumulate()`. See `DcsClustLAccumulate`
`ippsDcsClustLCompute()`. See `DcsClustLCompute`
`ippsDCTFwd()`. See `DCTFwd`
`ippsDCTFwdFree()`. See `DCTFwdFree`
`ippsDCTFwdGetBufSize()`. See `DCTFwdGetBufSize`
`ippsDCTFwdInitAlloc()`. See `DCTFwdInitAlloc`
`ippsDCTInv()`. See `DCTInv`
`ippsDCTInvFree()`. See `DCTInvFree`
`ippsDCTInvGetBufSize()`. See `DCTInvGetBufSize`
`ippsDCTInvInitAlloc()`. See `DCTInvInitAlloc`
`ippsDCTLifter()`. See `DCTLifter`
`ippsDCTLifterFree()`. See `DCTLifterFree`
`ippsDCTLifterGetSize_MulC0()`. See `DCTLifterGetSize_MulC0`
`ippsDCTLifterInit_MulC0()`. See `DCTLifterInit_MulC0`
`ippsDCTLifterInitAlloc()`. See `DCTLifterInitAlloc`
`ippsDecodeAdaptiveVector_G723()`. See `DecodeAdaptiveVector_G723`
`ippsDecodeAdaptiveVector_G729()`. See `DecodeAdaptiveVector_G729`
`ippsDecodeChanPairElt_AAC`, 10-122
`ippsDecodeChanPairElt_AAC()`. See `DecodeChanPairElt_AAC`
`ippsDecodeDatStrElt_AAC`, 10-127
`ippsDecodeDatStrElt_AAC()`. See `DecodeDatStrElt_AAC`
`ippsDecodeExtensionHeader_AAC`, 10-143
`ippsDecodeExtensionHeader_AAC()`. See `DecodeExtensionHeader_AAC`
`ippsDecodeFillElt_AAC`, 10-128
`ippsDecodeFillElt_AAC()`. See `DecodeFillElt_AAC`
`ippsDecodeGain_G729()`. See `DecodeGain_G729`
`ippsDecodeGIT()`. See `DecodeGIT`
`ippsDecodeGITGetSize()`. See `DecodeGITGetSize`
`ippsDecodeGITInit()`. See `DecodeGITInit`
`ippsDecodeGITInitAlloc()`. See `DecodeGITInitAlloc`
`ippsDecodeHuff()`. See `ippsDecodeHuff`
`ippsDecodeHuffInit()`. See `ippsDecodeHuffInit`
`ippsDecodeHuffInitAlloc()`. See `ippsDecodeHuffInitAlloc`
`ippsDecodeHuffOne()`. See `ippsDecodeHuffOne`
`ippsDecodeIsStereo_AAC()`. See `DecodeIsStereo_AAC`
`ippsDecodeIsStereo_AAC_32s`, 10-133
`ippsDecodeLZ77()`. See `DecodeLZ77`
`ippsDecodeLZ77DynamicHuff()`. See `DecodeLZ77DynamicHuff`
`ippsDecodeLZ77FixedHuff()`. See `DecodeLZ77FixedHuff`
`ippsDecodeLZ77GetBlockType()`. See `DecodeLZ77GetBlockType`
`ippsDecodeLZ77GetPairs()`. See `DecodeLZ77GetPairs`
`ippsDecodeLZ77GetSize()`. See `DecodeLZ77GetSize`
`ippsDecodeLZ77GetStatus()`. See `DecodeLZ77GetStatus`
`ippsDecodeLZ77Init()`. See `DecodeLZ77Init`
`ippsDecodeLZ77InitAlloc()`. See `DecodeLZ77InitAlloc`
`ippsDecodeLZ77Reset()`. See `DecodeLZ77Reset`
`ippsDecodeLZ77SetPairs()`. See `DecodeLZ77SetPairs`
`ippsDecodeLZ77SetStatus()`. See `DecodeLZ77SetStatus`
`ippsDecodeLZ77StoredBlock()`. See `DecodeLZ77StoredBlock`
`ippsDecodeLZSS()`. See `DecodeLZSS`
`ippsDecodeLZSSInit()`. See `DecodeLZSSInit`
`ippsDecodeLZSSInitAlloc()`. See `DecodeLZSSInitAlloc`
`ippsDecodeMainHeader_AAC`, 10-142
`ippsDecodeMainHeader_AAC()`. See `DecodeMainHeader_AAC`

ippsDecodeMsPNS_AAC(). See DecodeMsPNS_AAC
ippsDecodeMsStereo_AAC(). See
 DecodeMsStereo_AAC
ippsDecodeMsStereo_AAC_32s_I, 10-131
ippsDecodePNS_AAC(). See DecodePNS_AAC
ippsDecodePNS_AAC_32s, 10-145, 10-146
ippsDecodePrgCfgElt_AAC, 10-121
ippsDecodePrgCfgElt_AAC(). See
 DecodePrgCfgElt_AAC
ippsDecodeRLE(). See DecodeRLE
ippsDecodeTNS_AAC(). See DecodeTNS_AAC
ippsDecodeTNS_AAC_32s_I, 10-136
ippsDeinterleave(). See Deinterleave
ippsDeinterleaveSpectrum_AAC(). See
 DeinterleaveSpectrum_AAC
ippsDeinterleaveSpectrum_AAC_32s, 10-135
ippsDelta(). See Delta
ippsDeltaDelta(). See DeltaDelta
ippsDFTFree_C(). See DFTFree_C
ippsDFTFree_R(). See DFTFree_R
ippsDFTFwd_CToC(). See DFTFwd_CToC
ippsDFTFwd_RToCCS(). See DFTFwd_RToCCS
ippsDFTFwd_RToPack(). See DFTFwd_RToPack
ippsDFTFwd_RToPerm(). See DFTFwd_RToPerm
ippsDFTGetBufSize_R(). See DFTGetBufSize_R
ippsDFTInitAlloc_C(). See DFTInitAlloc_C
ippsDFTInitAlloc_R(). See DFTInitAlloc_R
ippsDFTInv_CCSToR(). See DFTInv_CCSToR
ippsDFTInv_CToC(). See DFTInv_CToC
ippsDFTInv_PackToR(). See DFTInv_PackToR
ippsDFTInv_PermToR(). See DFTInv_PermToR
ippsDFTOutOrdFree_C(). See DFTOutOrdFree_C
ippsDFTOutOrdFwd_CToC(). See
 DFTOutOrdFwd_CToC
ippsDFTOutOrdGetBufSize_C(). See
 DFTOutOrdGetBufSize_C
ippsDFTOutOrdInitAlloc_C(). See
 DFTOutOrdInitAlloc_C
ippsDFTOutOrdInv_CToC(). See DFTOutOrdInv_CToC
ippsDiv, 12-6

ippsDiv(). See Div
ippsDivC(). See DivC
ippsDivCRev(). See DivCRev
ippsDotProd(). See DotProd
ippsDotProd_G729(). See DotProd_G729
ippsDotProdAutoScale(). See DotProdAutoScale
ippsDotProdColumn(). See DotProdColumn
ippsDTW(). See DTW
ippsDurbin(). See Durbin
ippsEmptyFBankInitAlloc(). See EmptyFBankInitAlloc
ippsEncodeGIT. See EncodeGIT
ippsEncodeGITGetSize(). See EncodeGITGetSize
ippsEncodeGITInit(). See EncodeGITInit
ippsEncodeGITInitAlloc(). See EncodeGITInitAlloc
ippsEncodeHuff(). See EncodeHuff
ippsEncodeHuffFinal(). See EncodeHuffFinal
ippsEncodeHuffInit(). See EncodeHuffInit
ippsEncodeHuffInitAlloc(). See EncodeHuffInitAlloc
ippsEncodeHuffOne(). See EncodeHuffOne
ippsEncodeLZ77(). See EncodeLZ77
ippsEncodeLZ77DynamicHuff(). See
 EncodeLZ77DynamicHuff
ippsEncodeLZ77FixedHuff(). See
 EncodeLZ77FixedHuff
ippsEncodeLZ77Flush(). See EncodeLZ77Flush
ippsEncodeLZ77GetPairs(). See EncodeLZ77GetPairs
ippsEncodeLZ77GetSize(). See EncodeLZ77GetSize
ippsEncodeLZ77GetStatus(). See EncodeLZ77GetStatus
ippsEncodeLZ77Init(). See EncodeLZ77Init
ippsEncodeLZ77InitAlloc(). See EncodeLZ77InitAlloc
ippsEncodeLZ77Reset(). See EncodeLZ77Reset
ippsEncodeLZ77SelectHuffMode(). See
 EncodeLZ77SelectHuffMode
ippsEncodeLZ77SetPairs(). See EncodeLZ77SetPairs
ippsEncodeLZ77SetStatus(). See EncodeLZ77SetStatus
ippsEncodeLZ77StoredBlock(). See
 EncodeLZ77StoredBlock
ippsEncodeLZSS(). See EncodeLZSS
ippsEncodeLZSSFlush(). See EncodeLZSSFlush
ippsEncodeLZSSInit(). See EncodeLZSSInit

ippsEncodeLZSSInitAlloc(). See EncodeLZSSInitAlloc
ippsEncodeRLE(). See EncodeRLE
ippsEncodeTNS_AAC(). See EncodeTNS_AAC
ippsEncodeTNS_AAC_32s_I, 10-151
ippsEntropy(). See Entropy
ippsEqual(). See Equal
ippsErf, 12-54
ippsErfc, 12-56
ippSetDenormAreZeros, 3-11
ippSetFlushToZero, 3-10
ippSetNumThreads, 3-13
ippsEvalDelta(). See EvalDelta
ippsEvalDelta_Aurora(). See EvalDelta_Aurora
ippsEvalFBank(). See EvalFBank
ippsExp, 12-21
ippsExp(). See Exp
ippsExpNegSqr(). See ExpNegSqr
ippsFBankFree(). See FBankFree
ippsFBankGetCenters(). See FBankGetCenters
ippsFBankGetCoeffs(). See FBankGetCoeffs
ippsFBankSetCenters(). See FBankSetCenters
ippsFBankSetCoeffs(). See FBankSetCoeffs
ippsFDPFree(). See FDPFree
ippsFDPFwd(). See FDPFwd
ippsFDPGetSize(). See FDPGetSize
ippsFDPIInit(). See FDPInit
ippsFDPIInitAlloc(). See FDPInitAlloc
ippsFDPIInv(). See FDPInv
ippsFDPReset(). See FDPReset
ippsFDPResetGroup(). See FDPResetGroup
ippsFDPResetSfb(). See FDPResetSfb
ippsFFTFree_C(). See FFTFree_C
ippsFFTFree_R(). See FFTFree_R
ippsFFTFwd_CToC(). See FFTFwd_CToC
ippsFFTFwd_RToCCS(). See FFTFwd_RToCCS
ippsFFTFwd_RToPack(). See FFTFwd_RToPack
ippsFFTFwd_RToPerm(). See FFTFwd_RToPerm
ippsFFTGetBufSize_C(). See FFTGetBufSize_C
ippsFFTGetBufSize_R(). See FFTGetBufSize_R
ippsFFTGetSize_C(). See FFTGetSize_C
ippsFFTGetSize_R(). See FFTGetSize_R
ippsFFTInit_C(). See FFTInit_C
ippsFFTInit_R(). See FFTInit_R
ippsFFTInitAlloc_C(). See FFTInitAlloc_C
ippsFFTInitAlloc_R(). See FFTInitAlloc_R
ippsFFTInv_CCSToR(). See FFTInv_CCSToR
ippsFFTInv_CToC(). See FFTInv_CToC
ippsFFTInv_PackToR(). See FFTInv_PackToR
ippsFFTInv_PermToR(). See FFTInv_PermToR
ippsFillShortlist_Column(). See FillShortlist_Column
ippsFillShortlist_Row(). See FillShortlist_Row
ippsFilterAECNLMS(). See FilterAECNLMS
ippsFilterMedian. See FilterMedian
ippsFilterUpdateEMNS(). See FilterUpdateEMNS
ippsFilterUpdateWiener(). See FilterUpdateWiener
ippsFind(). See Find, FindRev
ippsFindC(). See FindC, FindRevC
ippsFindCAny(). See FindCAny, FindRevCAny
ippsFindNearest(). See FindNearest
ippsFindNearestOne(). See FindNearestOne
ippsFindPeaks(). See FindPeaks
ippsFindRevC(). See FindC, FindRevC
ippsFindRevCAny(). See FindCAny, FindRevCAny
ippsFIR(). See FIR
ippsFIR_Direct(). See FIR_Direct
ippsFIRBlockFree(). See FIRBlockFree
ippsFIRBlockInitAlloc(). See FIRBlockInitAlloc
ippsFIRBlockOne(). See FIRBlockOne
ippsFIRFree(). See FIRFree
ippsFIRGenBandpass(). See FIRGenBandpass
ippsFIRGenBandstop(). See FIRGenBandstop
ippsFIRGenHighpass(). See FIRGenHighpass
ippsFIRGenLowpass(). See FIRGenLowpass
ippsFIRGetDlyLine(). See FIRGetDlyLine
ippsFIRGetStateSize(). See FIRGetStateSize
ippsFIRGetTaps(). See FIRGetTaps
ippsFIRInit(). See FIRInit
ippsFIRInitAlloc(). See FIRInitAlloc

ippsFIRLMS(). See FIRLMS
ippsFIRLMSFree(). See FIRLMSFree
ippsFIRLMSGetDlyLine(). See FIRLMSGetDlyLine
ippsFIRLMSGetTaps(). See FIRLMSGetTaps
ippsFIRLMSInitAlloc(). See FIRLMSInitAlloc
ippsFIRLMSMRFree(). See FIRLMSMRFree
ippsFIRLMSMRGetDlyLine(). See
 FIRLMSMRGetDlyLine
ippsFIRLMSMRGetDlyVal(). See
 FIRLMSMRGetDlyVal
ippsFIRLMSMRGetTaps(). See FIRLMSMRGetTaps
ippsFIRLMSMRGetTapsPointer(). See
 FIRLMSMRGetTapsPointer
ippsFIRLMSMRInitAlloc(). See FIRLMSMRInitAlloc
ippsFIRLMSMROne(). See FIRLMSMROne
ippsFIRLMSMROneVal(). See FIRLMSMROneVal
ippsFIRLMSMRPutVal(). See FIRLMSMRPutVal
ippsFIRLMSMRSetDlyLine(). See
 FIRLMSMRSetDlyLine
ippsFIRLMSMRSetMu(). See FIRLMSMRSetMu
ippsFIRLMSMRSetTaps(). See FIRLMSMRSetTaps
ippsFIRLMSMRUpdateTaps(). See
 FIRLMSMRUpdateTaps
ippsFIRLMSOne_Direct(). See FIRLMSOne_Direct
ippsFIRLMSSetDlyLine(). See FIRLMSSetDlyLine
ippsFIRMR_Direct(). See FIRMR_Direct
ippsFIRMRGetStateSize(). See FIRMRGetStateSize
ippsFIRMRInit(). See FIRMRInit
ippsFIRMRInitAlloc(). See FIRMRInitAlloc
ippsFIROne(). See FIROne
ippsFIROne_Direct(). See FIROne_Direct
ippsFIRSetDlyLine(). See FIRSetDlyLine
ippsFIRSetTaps(). See FIRSetTaps
ippsFIRSparse(). See FIRSparse
ippsFIRSparseGetStateSize(). See
 FIRSparseGetStateSize
ippsFIRSparseInit(). See FIRSparseInit
ippsFixedCodebookSearch_G729(). See
 FixedCodebookSearch_G729
ippsFlip(). See Flip
ippsFormVector(). See FormVector
ippsFormVectorVQ(). See FormVectorVQ
ippsFree, 3-5
ippsGainControl_G723(). See GainControl_G723
ippsGainControl_G729(). See GainControl_G729
ippsGainQuant_G723(). See GainQuant_G723
ippsGainQuant_G729(). See GainQuant_G729
ippsGaussianDist(). See GaussianDist
ippsGaussianMerge(). See GaussianMerge
ippsGaussianSplit(). See GaussianSplit
ippsGetBufSize_C(). See DFTGetBufSize_C
ippsGetCdbkSize(). See GetCdbkSize
ippsGetCodebook(). See GetCodebook
ippsGetLibVersion(). See GetLibVersion
ippsGetSizeMCRA(). See GetSizeMCRA
ippsGetVarPointDV(). See GetVarPointDV
ippsGITFree(). See GITFree
ippsGoertz(). See Goertz
ippsGoertzTwo(). See GoertzTwo
ippsHarmonicFilter(). See HarmonicFilter
ippsHarmonicNoiseSubtract_G723(). See
 HarmonicNoiseSubtract_G723
ippsHarmonicSearch_G723(). See
 HarmonicSearch_G723
ippsHash(). See Hash
ippsHighBandCoding_Aurora(). See
 HighBandCoding_Aurora
ippsHighPassFilter_G723(). See HighPassFilter_G723
ippsHighPassFilter_G729(). See HighPassFilter_G729
ippsHighPassFilterInit_G729(). See
 HighPassFilterInit_G729
ippsHighPassFilterSize_G729(). See
 HighPassFilterSize_G729
ippsHilbert(). See Hilbert
ippsHilbertFree(). See HilbertFree
ippsHilbertInitAlloc(). See HilbertInitAlloc
ippsHuffFree(). See HuffFree
ippsHuffGetDstBuffSize(). See ippsHuffGetDstBuffSize
ippsHuffGetLenCodeTable(). See HuffGetLenCodeTable
ippsHuffGetSize(). See HuffGetSize

ippsHuffLenCodeTablePack(). See
 ippsHuffLenCodeTablePack

ippsHuffLenCodeTableUnpack(). See
 ippsHuffLenCodeTableUnpack

ippsHuffmanDecode_MP3(). See HuffmanDecode_MP3

ippsHuffmanEncode_MP3(). See HuffmanEncode_MP3

ippsHuffmanEncode_MP3_32s1u, 10-86

ippsIIR(). See IIR

ippsIIR_BiQuadDirect(). See IIR_BiQuadDirect

ippsIIR_Direct(). See IIR_Direct

ippsIIR16s_G723(). See IIR16s_G723

ippsIIR16s_G729(). See IIR16s_G729

ippsIIRFree(). See IIRFree

ippsIIRGetDlyLine(). See IIRGetDlyLine

ippsIIRGetStateSize(). See IIRGetStateSize

ippsIIRGetStateSize_BiQuad(). See
 IIRGetStateSize_BiQuad

ippsIIRInitAlloc(). See IIRInitAlloc

ippsIIRInitAlloc_BiQuad(). See IIRInitAlloc_BiQuad

ippsIIROne(). See IIROne

ippsIIROne_BiQuadDirect(). See IIROne_BiQuadDirect

ippsIIROne_Direct(). See IIROne_Direct

ippsIIRSetDlyLine(). See IIRSetDlyLine

ippsIIRSparse(). See IIRSparse

ippsIIRSparseGetStateSize(). See IIRSparseGetStateSize

ippsIIRSparseInit(). See IIRSparseInit

ippsImag(). See Imag

ippsIndexSelect_VQ(). See IndexSelect_VQ

ippsInitBitReservoir_MP3, 10-92

ippsInitMCRA(). See InitMCRA

ippsInsert(). See Insert

ippsInterleave(). See Interleave

ippsInterpolate_G729(). See Interpolate_G729

ippsInv, 12-4

ippsInvCbrrt, 12-13

ippsInvSqrt, 12-10

ippsInvSqrt(). See InvSqrt

ippsJoin(). See Join

ippsJoinScaled(). See JoinScaled

ippsJointStereoEncode_MP3(). See
 JointStereoEncode_MP3

ippsJointStereoEncode_MP3_32s_I, 10-75

ippsLagWindow_G729(). See LagWindow_G729

ippsLevinsonDurbin_G723(). See LevinsonDurbin_G723

ippsLevinsonDurbin_G729(). See LevinsonDurbin_G729

ippsLevinsonDurbin_GSMAMR, 9-127

ippsLinearPrediction(). See LinearPrediction

ippsLinearToMel(). See LinearToMel

ippsLinToALaw(). See LinToALaw

ippsLinToMuLaw(). See LinToMuLaw

ippsLn, 12-23

ippsLn(). See Ln

ippsLog10, 12-25

ippsLogAdd(). See LogAdd

ippsLogGauss(). See LogGauss

ippsLogGaussAdd(). See LogGaussAdd

ippsLogGaussAddMultiMix(). See
 LogGaussAddMultiMix

ippsLogGaussMax(). See LogGaussMax

ippsLogGaussMaxMultiMix(). See
 LogGaussMaxMultiMix

ippsLogGaussMixture(). See LogGaussMixture

ippsLogGaussMixtureSelect(). See
 LogGaussMixtureSelect

ippsLogGaussMultiMix(). See LogGaussMultiMix

ippsLogGaussSingle(). See LogGaussSingle

ippsLogSub(). See LogSub

ippsLogSum(). See LogSum

ippsLongTermPostFilter_G729(). See
 LongTermPostFilter_G729

ippsLongTermPredict_AAC(). See
 LongTermPredict_AAC

ippsLongTermPredict_AAC_32s, 10-152

ippsLongTermReconstruct_AAC(). See
 LongTermReconstruct_AAC

ippsLongTermReconstruct_AAC_32s, 10-149

ippsLowercase(). See Lowercase

ippsLowercaseLatin(). See Lowercase

ippsLowHighFilter_Aurora(). See

LowHighFilter_Aurora
ippsLPCToLSF_G723(). See LPCToLSF_G723
ippsLPCToLSP_G729(). See LPCToLSP_G729
ippsLPCToLSP_GSMAMR_16s, 9-129
ippsLPToCepstrum(). See LPToCepstrum
ippsLPToLSP(). See LPToLSP
ippsLPToReflection(). See LPToReflection
ippsLPToSpectrum(). See LPToSpectrum
ippsLSFDecode_G723(). See LSFDecode_G723
ippsLSFDecode_G729(). See LSFDecode_G729
ippsLSFDecodeErased_G729(). See
LSFDecodeErased_G729
ippsLSFQuant_G723(). See LSFQuant_G723
ippsLSFQuant_G729(). See LSFQuant_G729
ippsLSFToLPC_G723(). See LSFToLPC_G723
ippsLSFToLSP_G729(). See LSFToLSP_G729
ippsLShiftC(). See LShiftC
ippsLSPQuant_G729(). See LSPQuant_G729
ippsLSPQuant_GSMAMR_16s, 9-132
ippsLSPToLPC_G729(). See LSPToLPC_G729
ippsLSPToLPC_GSMAMR_16s, 9-130
ippsLSPToLSF_G729(). See LSPToLSF_G729
ippsLtpUpdate_AAC(). See LtpUpdate_AAC
ippsLtpUpdate_AAC_32s, 10-155
ippsLZ77Free(). See LZ77Free
ippsLZSSFree(). See LZSSFree
ippsLZSSGetSize(). See LZSSGetSize
ippsMagnitude(). See Magnitude
ippsMagSquared(). See MagSquared
ippsMahDist(). See MahDist
ippsMahDistMultiMix(). See MahDistMultiMix
ippsMahDistSingle(). See MahDistSingle
ippsMakeFloat(). See MakeFloat
ippsMalloc, 3-4
ippsMatVecMul(). See MatVecMul
ippsMax(). See Max
ippsMaxAbs(). See MaxAbs
ippsMaxAbsIndx(). See MaxAbsIndx
ippsMaxEvery(). See MaxEvery

ippsMaxIndx(). See MaxIndx
ippsMaxOrder(). See MaxOrder
ippsMDCTFwd(). See MDCTFwd, MDCTInv
ippsMDCTFwd_AAC(). See MDCTFwd_AAC
ippsMDCTFwd_AAC_32s, 10-150
ippsMDCTFwd_MP3(). See MDCTFwd_MP3
ippsMDCTFwd_MP3_32s, 10-68
ippsMDCTFwdFree(). See MDCTFwdFree,
MDCTInvFree
ippsMDCTFwdGetBufSize(). See
MDCTFwdGetBufSize, MDCTInvGetBufSize
ippsMDCTFwdGetSize(). See MDCTFwdGetSize,
MDCTInvGetSize
ippsMDCTFwdInit(). See MDCTFwdInit, MDCTInvInit
ippsMDCTFwdInitAlloc(). See MDCTFwdInitAlloc,
MDCTInvInitAlloc
ippsMDCTInv(). See MDCTFwd, MDCTInv
ippsMDCTInv_AAC(). See MDCTInv_AAC
ippsMDCTInv_AAC_32s16s, 10-140
ippsMDCTInv_MP3(). See MDCTInv_MP3
ippsMDCTInvFree(). See MDCTFwdFree,
MDCTInvFree
ippsMDCTInvGetBufSize(). See MDCTFwdGetBufSize,
MDCTInvGetBufSize
ippsMDCTInvGetSize(). See MDCTFwdGetSize,
MDCTInvGetSize
ippsMDCTInvInit(). See MDCTFwdInit, MDCTInvInit
ippsMDCTInvInitAlloc(). See MDCTFwdInitAlloc,
MDCTInvInitAlloc
ippsMean(). See Mean
ippsMeanColumn(). See MeanColumn
ippsMeanVarAcc(). See MeanVarAcc
ippsMeanVarColumn(). See MeanVarColumn
ippsMelFBankGetSize(). See MelFBankGetSize
ippsMelFBankInit(). See MelFBankInit
ippsMelFBankInitAlloc(). See MelFBankInitAlloc
ippsMelFBankInitAlloc_Aurora(). See
MelFBankInitAlloc_Aurora
ippsMelLinFBankInitAlloc(). See MelLinFBankInitAlloc
ippsMelToLinear(). See MelToLinear

-
- ippsMin(). See Min
 - ippsMinAbs(). See MinAbs
 - ippsMinAbsIndx(). See MinAbsIndx
 - ippsMinEvery(). See MinEvery
 - ippsMinIndxt(). See MinIndx
 - ippsMinMax(). See MinMax
 - ippsMinMaxIndx(). See MinMaxIndx
 - ippsMove(). See Move
 - ippsMPMLQFixedCodebookSearch_G723(). See MPMLQFixedCodebookSearch_G723
 - ippsMTFFree(). See MTFFree
 - ippsMTFFwd(). See MTFFwd
 - ippsMTFGetSize(). See MTFGetSize
 - ippsMTFInit(). See MTFInit
 - ippsMTFInitAlloc(). See MTFInitAlloc
 - ippsMTFInv(). See MTFInv
 - ippsMul(). See Mul
 - ippsMul_NR(). See Mul_NR
 - ippsMuLawToALaw(). See MuLawToALaw
 - ippsMuLawToLin(). See MuLawToLin
 - ippsMulC_NR(). See MulC_NR
 - ippsMulColumn(). See MulColumn
 - ippsMulPack(). See MulPack
 - ippsMulPackConj(). See MulPackConj
 - ippsMulPerm(). See MulPerm
 - ippsMulPowerC_NR(). See MulPowerC_NR
 - ippsNewVar(). See NewVar
 - ippsNoiselessDecode_AAC, 10-153
 - ippsNoiselessDecode_AAC(). See NoiselessDecode_AAC
 - ippsNoiselessDecoder_LC_AAC, 10-123
 - ippsNoiselessDecoder_LC_AAC(). See NoiselessDecoder_LC_AAC
 - ippsNoiseSpectrumUpdate_Aurora(). See NoiseSpectrumUpdate_Aurora
 - ippsNorm(). See Norm
 - ippsNormalize(). See Normalize
 - ippsNormalizeColumn(). See NormalizeColumn
 - ippsNormalizeInRange(). See NormalizeInRange
 - ippsNormDiff(). See NormDiff
 - ippsNormEnergy(). See NormEnergy
 - ippsNot(). See Not
 - ippsNthMaxElement(). See NthMaxElement
 - ippsOpenLoopPitchSearch_G723(). See OpenLoopPitchSearch_G723
 - ippsOpenLoopPitchSearch_G729(). See OpenLoopPitchSearch_G729
 - ippsOr(). See Or
 - ippsOrC(). See OrC
 - ippsOutProbPreCalc(). See OutProbPreCalc
 - ippsPackFrameHeader_MP3, 10-89
 - ippsPackFrameHeader_MP3(). See PackFrameHeader_MP3
 - ippsPackScalefactors_MP3(). See PackScalefactors_MP3
 - ippsPackScalefactors_MP3_8s1u, 10-84
 - ippsPackSideInfo_MP3, 10-90
 - ippsPackSideInfo_MP3(). See PackSideInfo_MP3
 - ippsPeriodicity(). See Periodicity
 - ippsPeriodicityLSPE(). See PeriodicityLSPE
 - ippsPhase(). See Phase, complex
 - ippsPitchmarkToF0(). See PitchmarkToF0
 - ippsPitchPostFilter_G723(). See PitchPostFilter_G723
 - ippsPolarToCart(). See PolarToCart, complex
 - ippsPow, 12-15
 - ippsPow34(). See Pow34
 - ippsPow43(). See Pow43
 - ippsPowerSpectr(). See PowerSpectr, complex
 - ippsPowx, 12-18
 - ippsPredictCoef_SBR(). See PredictCoef_SBR
 - ippsPreemphasize(). See Preemphasize
 - ippsPreemphasize_G729(). See Preemphasize_G729
 - ippsPreSelect_VQ(). See PreSelect_VQ
 - ippsPsych_MP3_16s, 10-70
 - ippsPsychoacousticModelTwo_MP3(). See PsychoacousticModelTwo_MP3
 - ippsQMFSynthesisInit_SBR(). See QMFSynthesisInit_SBR
 - ippsQRTransColumn(). See QRTransColumn
 - ippsQuantInv_AAC(). See QuantInv_AAC
 - ippsQuantInv_AAC_32s_I, 10-129

ippsQuantize_MP3(). See Quantize_MP3
ippsQuantize_MP3_32s_I, 10-78
ippsQuantLSPDecode_GSMAMR_16s, 9-135
ippsRandGauss(). See RandGauss
ippsRandGauss_Direct(). See RandGauss_Direct
ippsRandGaussFree(). See RandGaussFree
ippsRandGaussGetSize(). See RandGaussGetSize
ippsRandGaussInit(). See RandGaussInit
ippsRandGaussInitAlloc(). See RandGaussInitAlloc
ippsRandomNoiseExcitation_G729B(). See
 RandomNoiseExcitation_G729B
ippsRandUniform_Direct(). See RandUniform_Direct
ippsRandUniformFree(). See RandUniformFree
ippsRandUniformGetSize(). See RandUniformGetSize
ippsRandUniformInit(). See RandUniformInit
ippsRandUniformInitAlloc(). See RandUniformInitAlloc
ippsRandUnifrom(). See RandUnifrom
ippsReal(). See Real
ippsRealToCplx(). See RealToCplx
ippsRecSqrt(). See RecSqrt
ippsReflectionToAR(). See ReflectionToAR
ippsReflectionToLP(). See ReflectionToLP
ippsReflectionToTilt(). See ReflectionToTilt
ippsRegExpFind(). See RegExpFind
ippsRegExpFree(). See RegExpFree
ippsRegExpGetSize(). See RegExpGetSize
ippsRegExpInit(). See RegExpInit
ippsRegExpInitAlloc(). See RegExpInitAlloc
ippsRegExpSetMatchLimit(). See RegExpSetMatchLimit
ippsRemove(). See Remove
ippsReplaceC(). See ReplaceC
ippsReQuantize_MP3(). See ReQuantize_MP3
ippsResamplePolyphase(). See ResamplePolyphase
ippsResamplePolyphaseFree(). See
 ResamplePolyphaseFree
ippsResamplePolyphaseGetFilter(). See
 ResamplePolyphaseGetFilter
ippsResamplePolyphaseGetSize(). See
 ResamplePolyphaseGetSize
ippsResamplePolyphaseInit(). See
 ResamplePolyphaseInit
ippsResamplePolyphaseInitAlloc(). See
 ResamplePolyphaseInitAlloc
ippsResamplePolyphaseSetFilter(). See
 ResamplePolyphaseSetFilter
ippsResidualFilter_Aurora(). See ResidualFilter_Aurora
ippsResidualFilter_G729(). See ResidualFilter_G729
ippsRShiftC(). See RShiftC
ippsSampleDown(). See SampleDown
ippsSampleUp(). See SampleUp
ippsScale(). See Scale
ippsScaleLM(). See ScaleLM
ippsSchur(). See Schur
ippsSet(). See Set
ippsShortTermPostFilter_G729(). See
 ShortTermPostFilter_G729
ippsSignChangeRate(). See SignChangeRate
ippsSin, 12-29
ippsSinC(). See SinC
ippsSinCos, 12-30
ippsSinh, 12-45
ippsSmoothedPowerSpectrumAurora(). See
 SmoothedPowerSpectrumAurora
ippsSortAscend(). See SortAscend
ippsSortDescend(). See SortDescend
ippsSplitC(). See SplitC
ippsSplitScaled(). See SplitScaled
ippsSplitVQ(). See SplitVQ
ippsSpread(). See Spread
ippsSqr(). See Sqr
ippsSqrt, 12-8
ippsSqrt(). See Sqrt
ippsStdDev(). See StdDev
ippsStepSizeUpdateAECNLMS(). See
 StepSizeUpdateAECNLMS
ippsSub(). See Sub
ippsSubC(). See SubC
ippsSubCRev(). See SubCRev
ippsSubRow(). See SubRow
ippsSum(). See Sum

ippsSumColumn(). See SumColumn
ippsSumColumnAbs(). See SumColumnAbs
ippsSumColumnSqr(). See SumColumnSqr
ippsSumLn(). See SumLn
ippsSumMeanVar(). See SumMeanVar
ippsSumRow(). See SumRow
ippsSumRowAbs(). See SumRowAbs
ippsSumRowSqr(). See SumRowSqr
ippsSVD(). See SVD
ippsSwapBytes(). See SwapBytes
ippsSynthesisDownFilter_SBR(). See
 SynthesisDownFilter_SBR
ippsSynthesisDownFilterGetSize_SBR(). See
 SynthesisDownFilterGetSize_SBR
ippsSynthesisDownFilterInit_SBR(). See
 SynthesisDownFilterInit_SBR
ippsSynthesisFilter_G723(). See SynthesisFilter_G723
ippsSynthesisFilter_G729(). See SynthesisFilter_G729
ippsSynthesisFilter_SBR(). See SynthesisFilter_SBR
ippsSynthesisFilterGetSize_SBR(). See
 SynthesisFilterGetSize_SBR
ippsSynthPQMF_MP3(). See SynthPQMF_MP3
ippsTabsCalculation_Aurora(). See
 TabsCalculation_Aurora
ippsTan, 12-32
ippsTanh, 12-47
ippStaticInit, 3-16
ippStaticInitCpu, 3-17
ippsThreshold(). See Threshold
ippsThreshold_GT(). See Threshold_GT
ippsThreshold_GTAbs(). See Threshold_GTAbs
ippsThreshold_GTVal(). See Threshold_GTVal
ippsThreshold_LT(). See Threshold_LT
ippsThreshold_LTAbs(). See Threshold_LTAbs
ippsThreshold_LTInv(). See Threshold_LTInv
ippsThreshold_LTVal(). See Threshold_LTVal
ippsThreshold_LTValGTVal(). See
 Threshold_LTValGTVal
ippsTiltCompensation_G723(). See
 TiltCompensation_G723
ippsTiltCompensation_G729(). See
 TiltCompensation_G729
ippsToeplizMatrix_G723(). See ToeplizMatrix_G723
ippsToeplizMatrix_G729(). See ToeplizMatrix_G729
ippsTone_Direct(). See Tone_Direct
ippsToneFree(). See ToneFree
ippsToneGetStateSizeQ15(). See ToneGetStateSizeQ15
ippsToneInitAllocQ15(). See ToneInitAllocQ15
ippsToneInitQ15(). See ToneInitQ15
ippsToneQ15(). See ToneQ15
ippsTriangle_Direct(). See Triangle_Direct
ippsTriangleFree(). See TriangleFree
ippsTriangleGetStateSizeQ15(). See
 TriangleGetStateSizeQ15
ippsTriangleInitAllocQ15(). See TriangleInitAllocQ15
ippsTriangleInitQ15(). See TriangleInitQ15
ippsTriangleQ15(). See TriangleQ15
ippsTrimC(). See TrimC
ippsTrimCAny(). See TrimCAny
ippsTrimEndCAny(). See TrimCAny
ippsTrimStartCAny(). See TrimCAny
ippsUnitCurve(). See UnitCurve
ippsUnpackADIFHeader_AAC, 10-119
ippsUnpackADIFHeader_AAC(). See
 UnpackADIFHeader_AAC
ippsUnpackADTSFrameHeader_AAC, 10-120
ippsUnpackADTSFrameHeader_AAC(). See
 UnpackADTSFrameHeader_AAC
ippsUnpackFrameHeader_MP3(). See
 UnpackFrameHeader_MP3
ippsUnpackScaleFactors_MP3(). See
 UnpackScaleFactors_MP3
ippsUnpackSideInfo_MP3(). See UnpackSideInfo_MP3
ippsUpdateGConst(). See UpdateGConst
ippsUpdateLinear(). See UpdateLinear
ippsUpdateMean(). See UpdateMean
ippsUpdateNoisePSDMCRA(). See
 UpdateNoisePSDMCRA
ippsUpdatePathMetricsDV(). See UpdatePathMetricsDV
ippsUpdatePower(). See UpdatePower

ippsUpdateVar(). See UpdateVar
ippsUpdateWeight(). See UpdateWeight
ippsUppercase(). See Uppercase
ippsUppercaseLatin(). See Uppercase
ippsVADDecision_Aurora(). See VADDecision_Aurora
ippsVADFlush_Aurora(). See VADFlush_Aurora
ippsVADGetBufSize_Aurora(). See
 VADGetBufSize_Aurora
ippsVADInit_Aurora(). See VADInit_Aurora
ippsVarColumn(). See VarColumn
ippsVecMatMul(). See VecMatMul
ippsVectorJaehne(). See VectorJaehne
ippsVectorRamp(). See VectorRamp
ippsVLCCountBits(). See VLCCountBits
ippsVLCCountEscBits_AAC(). See
 VLCCountEscBits_AAC
ippsVLCCountEscBits_MP3(). See
 VLCCountEscBits_MP3
ippsVLCDecodeBlock(). See VLCDecodeBlock
ippsVLCDecodeEscBlock_AAC(). See
 VLCDecodeEscBlock_AAC
ippsVLCDecodeEscBlock_MP3(). See
 VLCDecodeEscBlock_MP3
ippsVLCDecodeFree(). See VLCDecodeFree
ippsVLCDecodeGetSize(). See VLCDecodeGetSize
ippsVLCDecodeInit(). See VLCDecodeInit
ippsVLCDecodeInitAlloc(). See VLCDecodeInitAlloc
ippsVLCDecodeOne(). See VLCDecodeOne
ippsVLCEncodeBlock(). See VLCEncodeBlock
ippsVLCEncodeEscBlock_AAC(). See
 VLCEncodeEscBlock_AAC
ippsVLCEncodeEscBlock_MP3(). See
 VLCEncodeEscBlock_MP3
ippsVLCEncodeFree(). See VLCEncodeFree
ippsVLCEncodeGetSize(). See VLCEncodeGetSize
ippsVLCEncodeInit(). See VLCEncodeInit
ippsVLCEncodeInitAlloc(). See VLCEncodeInitAlloc
ippsVLCEncodeOne(). See VLCEncodeOne
ippsVQ(). See VQ
ippsVQCodeBookFree(). See VQCodeBookFree
ippsVQCodeBookGetSize(). See VQCodeBookGetSize
ippsVQCodeBookInit(). See VQCodeBookInit
ippsVQCodeBookInitAlloc(). See
 VQCodeBookInitAlloc
ippsVQMainSelect(). See VQMainSelect
ippsVQrReconstruction(). See VQReconstruction
ippsVQSingle_Sort(). See VQSingle_Sort
ippsVQSingle_Thresh(). See VQSingle_Thresh
ippsWaveProcessing_Aurora(). See
 WaveProcessing_Aurora
ippsWeightedMeanColumn(). See
 WeightedMeanColumn
ippsWeightedMeanVarColumn(). See
 WeightedMeanVarColumn
ippsWeightedSum(). See WeightedSum
ippsWeightedVarColumn(). See WeightedVarColumn
ippsWienerFilterDesign_Aurora(). See
 WienerFilterDesign_Aurora
ippsWinBartlett(). See WinBartlett
ippsWinBlackman(). See WinBlackman
ippsWinHamming(). See WinHamming
ippsWinHann(). See WinHann
ippsWinKaiser(). See WinKaiser
ippsWTFwd(). See WTFwd
ippsWTFwdFree(). See WTFwdFree
ippsWTFwdGetDlyLine(). See WTFwdGetDlyLine
ippsWTFwdInitAlloc(). See WTFwdInitAlloc
ippsWTFwdSetDlyLine(). See WTFwdSetDlyLine
ippsWTHaarFwd(). See WTHaarFwd
ippsWTHaarInv(). See WTHaarInv
ippsWTInv(). See WTInv
ippsWTInvFree(). See WTInvFree
ippsWTInvGetDlyLine(). See WTInvGetDlyLine
ippsWTInvInitAlloc(). See WTInvInitAlloc
ippsWTInvSetDlyLine(). See WTInvSetDlyLine
ippsXor(). See Xor
ippsXorC(). See XorC
ippsZero(). See Zero
ippsZeroMean(). See ZeroMean
ISFQuant_AMRWB, 9-206

ISFQuantDecode_AMRWB, 9-208
 ISFQuantDecodeDTX_AMRWB, 9-210
 ISFQuantDTX_AMRWB, 9-209
 ISFToISP_AMRWB, 9-181
 ISPToISF_Norm_AMRWB, 9-180
 ISPToLPC_AMRWB, 9-178

J

Join, 5-59
 JoinScaled, 5-60
 JointStereoEncode_MP3, 10-75

L

LagWindow_G729, 9-41
 LevinsonDurbin_G723, 9-94
 LevinsonDurbin_G728, 9-255
 LevinsonDurbin_G729, 9-28
 LevinsonDurbin_GSMAMR, 9-127
 line spectral frequencies, 9-32
 linear prediction analysis, 9-27
 linear predictors, 10-25
 LinearPrediction, 8-26
 LinearToMel, 8-46
 LinToALaw, 9-291
 LinToMuLaw, 9-289
 Ln, 5-48, 12-23
 Log10, 12-25
 LogAdd, 8-96
 LogGauss, 8-107
 LogGaussAdd, 8-120
 LogGaussAddMultiMix, 8-123
 LogGaussMax, 8-113
 LogGaussMaxMultiMix, 8-117
 LogGaussMixture, 8-125
 LogGaussMixtureSelect, 8-129
 LogGaussMultiMix, 8-111
 LogGaussSingle, 8-104
 Logical and shift functions, 5-5 thru 5-15

And, 5-7
 AndC, 5-6
 LShiftC, 5-13
 Not, 5-12
 Or, 5-9
 OrC, 5-8
 RShiftC, 5-14
 Xor, 5-11
 XorC, 5-10
 LogSub, 8-97
 LogSum, 8-99
 LongTermPostFilter_G729, 9-70
 LongTermPredict_AAC, 10-152
 LongTermReconstruct_AAC, 10-149
 Lowercase, 11-19
 LowercaseLatin, 11-19
 LowHighFilter_Aurora, 8-225
 LP coefficients, 9-36
 LPCInverseFilter_G728, 9-249
 LPCToISP_AMRWB, 9-177
 LPCToLSF_G723, 9-96
 LPCToLSP_G729, 9-30
 LPCToLSP_GSMAMR, 9-129
 LPToCepstrum, 8-32
 LPToLSP, 8-41
 LPToReflection, 8-34
 LPToSpectrum, 8-31
 LSFDecode_G723, 9-98
 LSFDecode_G729, 9-34
 LSFDecodeErased_G729, 9-35
 LSFQuant_G723, 9-99
 LSFQuant_G729, 9-32
 LSFToLPC_G723, 9-97
 LSFToLSP_G729, 9-32
 LSFToLSP_GSMAMR, 9-132
 LShiftC, 5-13
 LSP coefficients, 9-41
 LSPQuant_G729, 9-37
 LSPQuant_GSMAMR, 9-132
 LSPToLPC_G729, 9-36

LSPToLPC_GSMAMR, 9-130
LSPToLSF_G729, 9-40
LtpUpdate_AAC, 10-155
LZ77 dictionary-based compression, 13-45
LZ77Free, 13-53
LZSS compression, 13-33 thru ??
 EncodeLZSSInitAlloc, 13-34
LZSS compression algorithm, 13-33
LZSSFree, 13-34
LZSSGetSize, 13-36

M

MA predictor, 9-33
Magnitude, 5-65
MagSquared, 5-67
MahDist, 8-101
MahDistMultiMix, 8-102
MahDistSingle, 8-100
MakeFloat, 10-16
mantissa conversion and scaling functions, 10-14
manual organization, 1-5
MatVecMul, 8-20
Max, 5-117
MaxAbs, 5-119
MaxAbsIndx, 5-120
MaxEvery, 5-139
MaxIndx, 5-118
MaxOrder, 5-94
MDCTFwd, MDCTInv, 10-23
MDCTFwd_AAC, 10-150
MDCTFwd_MP3, 10-68
MDCTFwdFree, MDCTInvFree, 10-20
MDCTFwdGetBufSize, MDCTInvGetBufSize, 10-22
MDCTFwdGetSize, MDCTInvGetSize, 10-21
MDCTFwdInit, MDCTInvInit, 10-18
MDCTFwdInitAlloc, MDCTInvInitAlloc, 10-17
MDCTInv_AAC, 10-140
MDCTInv_MP3, 10-103
Mean, 5-127

MeanColumn, 8-142
MeanVarAcc, 8-156
MeanVarColumn, 8-145
Median filter functions, 6-126 thru 6-127
 FilterMedian, 6-126
MelFBankGetSize, 8-48
MelFBankInit, 8-49
MelFBankInitAlloc, 8-50
MelFBankInitAlloc_Aurora, 8-220
MelLinFBankInitAlloc, 8-53
MelToLinear, 8-45
memory allocation functions, 3-4 thru 3-6
 ippsFree, 3-5
 ippsMalloc, 3-4
Min, 5-120
MinAbs, 5-123
MinAbsIndx, 5-124
MinEvery, 5-139
MinIndx, 5-122
MinMax, 5-125
MinMaxIndx, 5-126
MMX technology, 1-1
Model adaptation functions, 8-175 thru 8-187
 AddMulColumn, 8-175
 AddMulRow, 8-176
 DotProdColumn, 8-178
 MulColumn, 8-179
 QRTransColumn, 8-177
 SumColumnAbs, 8-180
 SumColumnSqr, 8-181
 SumRowAbs, 8-182
 SumRowSqr, 8-183
 SVD, SVDSort, 8-184
 WeightedSum, 8-186
Model estimation functions, 8-141 thru 8-174
 BhatDist, 8-164
 DcsClustLAccumulate, 8-172
 DcsClustLCompute, 8-173
 Entropy, 8-161
 ExpNegSqr, 8-163
 GaussianDist, 8-157
 GaussianMerge, 8-160

- GaussianSplit, 8-159
- MeanColumn, 8-142
- MeanVarAcc, 8-156
- MeanVarColumn, 8-145
- NormalizeColumn, 8-152
- NormalizeInRange, 8-154
- OutProbPreCalc, 8-171
- SinC, 8-162
- UpdateGConst, 8-169
- UpdateMean, 8-166
- UpdateVar, 8-167
- UpdateWeight, 8-168
- VarColumn, 8-143
- WeightedMeanColumn, 8-146
- WeightedMeanVarColumn, 8-149
- WeightedVarColumn, 8-148
- Model evaluation functions, 8-94 thru 8-141
 - AddNRows, 8-94
 - BuildSignTable, 8-132
 - DTW, 8-139
 - FillShortlist_Column, 8-136
 - FillShortlist_Row, 8-134
 - LogAdd, 8-96
 - LogGauss, 8-107
 - LogGaussAdd, 8-120
 - LogGaussAddMultiMix, 8-123
 - LogGaussMax, 8-113
 - LogGaussMaxMultiMix, 8-117
 - LogGaussMixture, 8-125
 - LogGaussMixtureSelect, 8-129
 - LogGaussMultiMix, 8-111
 - LogGaussSingle, 8-104
 - LogSub, 8-97
 - LogSum, 8-99
 - MahDist, 8-101
 - MahDistMultiMix, 8-102
 - MahDistSingle, 8-100
 - ScaleLM, 8-95
- modified discrete cosine transform (MDCT), 10-17
- Move, 4-5
- Move To Front data transform method, 13-115
- move to front data transform. See MTF data transform
- MP3 audio decoder, 10-94
- MP3 audio decoder functions, 10-94 thru 10-107
 - HuffmanDecode_MP3, 10-99
 - MDCTInv_MP3, 10-103
 - ReQuantize_MP3, 10-101
 - SynthPQMF_MP3, 10-105
 - UnpackFrameHeader_MP3, 10-95
 - UnpackScaleFactors_MP3, 10-97
 - UnpackSideInfo_MP3, 10-96
- MP3 audio encoder
 - bit reservoir structure, 10-64
 - psychoacoustic model, 10-62
 - psychoacoustic model structure, 10-62
- MP3 audio encoder API
 - enumerated types, 10-65
- MP3, data structures, 10-60
- MP3, macro and constant definitions, 10-59
- MPEG-2
 - global macros, 10-108
- MPMLQFixedCodebookSearch_G723, 9-104
- MTF data transform, 13-115 thru ??
 - MTFFree, 13-117
 - MTFGetSize, 13-118
 - MTFInit, 13-117
 - MTFInitAlloc, 13-116
- MTFFree, 13-117
- MTFFwd, 13-118
- MTFGetSize, 13-118
- MTFInit, 13-117
- MTFInitAlloc, 13-116
- MTFInv, 13-119
- Mul, 5-24
- Mul_NR, 9-8
- MuLawToALaw, 9-293
- MuLawToLin, 9-288
- MulC, 5-22
- MulC_NR, 9-9
- MulColumn, 8-179
- MulPack, 7-13
- MulPackConj, 7-17
- MulPerm, 7-13, 7-15
- MulPowerC_NR, 9-11
- Multiplication of packed data, 7-12 thru 7-17
 - MulPack, 7-13

- MulPackConj, 7-17
- MulPerm, 7-13, 7-15
- multi-rate FIR LMS filter functions
 - FIRLMSMRFree, 6-78
 - FIRLMSMRGetDlyLine, 6-84
 - FIRLMSMRGetDlyVal, 6-86
 - FIRLMSMRGetTaps, 6-81
 - FIRLMSMRGetTapsPointer, 6-83
 - FIRLMSMRInitAlloc, 6-77
 - FIRLMSMROne, 6-88
 - FIRLMSMROneVal, 6-89
 - FIRLMSMRPutVal, 6-87
 - FIRLMSMRSetDlyLine, 6-85
 - FIRLMSMRSetMu, 6-79
 - FIRLMSMRSetTaps, 6-82
 - FIRLMSMRUpdateTaps, 6-80

N

- NewVar, 8-73
- NoiselessDecode_AAC, 10-153
- NoiselessDecoder_LC_AAC, 10-123
- NoiseSpectrumUpdate_Aurora, 8-217
- Norm, 5-131
- Normalize, 5-53
- NormalizeColumn, 8-152
- NormalizeInRange, 8-154
- NormDiff, 5-133
- NormEnergy, 8-69
- Not, 5-12
- notational conventions, 1-7
- NthMaxElement, 8-18

O

- online version, 1-7
- OpenLoopPitchSearch_AMRWB, 9-182
- OpenLoopPitchSearch_G723, 9-100
- OpenLoopPitchSearch_G729, 9-43
- OpenLoopPitchSearchDTXVAD1_GSMAMR, 9-141
- OpenLoopPitchSearchDTXVAD2_GSMAMR, 9-144
- OpenLoopPitchSearchNonDTX_GSMAMR, 9-139

- Or, 5-9
- OrC, 5-8
- out-of-order DFT, 7-40
- OutProbPreCalc, 8-171

P

- packed format for Fourier transform, 7-5
 - CCS format, 7-6
 - pack format, 7-5
 - perm format, 7-5
- PackFrameHeader_MP3, 10-89
- PackScalefactors_MP3, 10-84
- PackSideInfo_MP3, 10-90
- parallelism, 1-1
- Periodicity, 8-276
- PeriodicityLSPE, 8-275
- Phase, complex, 5-68
- PhaseDispersion_G729D, 9-84
- PhaseDispersionGetStateSize_G729D, 9-82
- PhaseDispersionInit_G729D, 9-82
- PhaseDispersionUpdate_G729D, 9-83
- pitch, 9-43
 - close loop, 9-103
 - open loop, 9-100
- pitch super resolution functions
 - CrossCorrCoeff, 8-90
 - CrossCorrCoeffDecim, 8-89
 - CrossCorrCoeffInterpolation, 8-92
- PitchmarkToF0, 8-39
- PitchPeriodExtraction_G728, 9-250
- PitchPostFilter_G723, 9-118
- platforms supported, 1-2
- PolarToCart, complex, 5-92
- polynomial coefficients, 9-96
- Polyphase functions
 - ResamplePolyphase, 8-213
 - ResamplePolyphaseFree, 8-212
 - ResamplePolyphaseGetFilter, 8-207
 - ResamplePolyphaseGetSize, 8-205
 - ResamplePolyphaseInit, 8-204
 - ResamplePolyphaseInitAlloc, 8-210

- ResamplePolyphaseSetFilter, 8-206
- post filter, 9-119
- PostFilter_G728, 9-246
- PostFilter_GSMAMR, 9-171
- PostFilterAdapterGetStateSize_G728, 9-247
- PostFilterAdapterStateInit_G728, 9-248
- PostFilterGetStateSize_G728, 9-245
- PostFilterInit_G728, 9-245
- Pow, 12-15
- Pow34, 10-10
- Pow43, 10-12
- power weighting, 9-11
- PowerSpectr, complex, 5-69
- Powx, 12-18
- PredictCoef_SBR, 10-175
- prediction error, 9-62
- prediction, in frequency domain, 10-29
- Preemphasize, 5-95
- Preemphasize_AMRWB, 9-187
- Preemphasize_G729, 9-85
- Preemphasize_GSMAMR, 9-160
- Preemphasize_GSMFR, 9-225
- prefix, in function names, 1-8
- prequantization, 10-10
- PreSelect_VQ, 10-51
- PsychoacousticModelTwo_MP3, 10-70

Q

- QMFSynthesisInit_SBR, 10-163
- QRTransColumn, 8-177
- QuantInv_AAC, 10-129
- quantization
 - in speech codecs, 9-33
 - inverse, 9-98
 - process, 9-38
- Quantize_MP3, 10-78
- QuantLSPDecode_GSMAMR, 9-135

R

- RandGauss, 4-35
- RandGauss_Direct, 4-36
- RandGaussFree, 4-32
- RandGaussGetSize, 4-33
- RandGaussInit, 4-34
- RandGaussInitAlloc, 4-31
- RandomNoiseExcitation_G729B, 9-88
- RandUniform_Direct, 4-30
- RandUniformFree, 4-27
- RandUniformGetSize, 4-29
- RandUniformInit, 4-28
- RandUniformInitAlloc, 4-26
- RandUnifrom, 4-29
- Real, 5-70
- RealToCplx, 5-72
- RecSqrt, 8-74
- ReflectionToAR, 8-36
- ReflectionToLP, 8-35
- ReflectionToTilt, 8-38
- RegExpFind, 11-31
- RegExpFree, 11-27
- RegExpGetSize, 11-29
- RegExpInit, 11-28
- RegExpInitAlloc, 11-26
- RegExpSetMatchLimit, 11-30
- regular expressions, 11-25
- related publications, 1-7
- Remove, 11-9
- ReplaceC, 11-17
- ReQuantize_MP3, 10-101
- ResamplePolyphase, 8-213
- ResamplePolyphaseFree, 8-212
- ResamplePolyphaseGetFilter, 8-207
- ResamplePolyphaseGetSize, 8-205
- ResamplePolyphaseInit, 8-204
- ResamplePolyphaseInitAlloc, 8-210
- ResamplePolyphaseSetFilter, 8-206
- residual signal, 9-67

ResidualFilter_AMRWB, 9-176
ResidualFilter_Aurora, 8-222
ResidualFilter_G729, 9-67
RFC1950, 13-45, 13-59, 13-63, 13-66, 13-68
RFC1951, 13-45, 13-54, 13-56, 13-57, 13-59, 13-63,
13-66, 13-68, 13-70, 13-76
RFC1952, 13-45, 13-59, 13-63, 13-66, 13-68
Rice-Colomb coding, 13-100
RLE. See run length encoding
rounding mode, in speech codec functions, 9-2
RPEQuantDecode_GSMFR, 9-218
RShiftC, 5-14
run length encoding (RLE), 13-120 thru ??
 DecodeRLE, 13-121
 EncodeRLE, 13-120

S

SampleDown, 5-142
SampleUp, 5-140
Sampling functions, 5-140 thru 5-144
 SampleDown, 5-142
 SampleUp, 5-140
Scale, 10-15
scale factor bands, 10-14
scale factors calculation, 10-13
ScaleLM, 8-95
Schur, 8-29
Schur_GSMFR, 9-223
Set, 4-6
SetDenormAreZeros(). See ippSetDenormAreZeros
SetFlushToZero(). See ippSetFlushToZero
SetNumThreads(). See ippSetNumThreads
Shift functions, 5-13
ShortTermAnalysisFilter_GSMFR, 9-220
ShortTermPostFilter_G729, 9-73
ShortTermSynthesisFilter_GSMFR, 9-221
signal name conventions, 1-8
SignChangeRate, 8-25
SIMD instructions, 1-1
Sin, 12-29
SinC, 8-162
SinCos, 12-30
single-rate FIR LMS filter functions
 FIRLMS, 6-71
 FIRLMSFree, 6-68
 FIRLMSGetDlyLine, 6-69
 FIRLMSGetTaps, 6-68
 FIRLMSInitAlloc, 6-67
 FIRLMSSetDlyLine, 6-74
 FIRLMSSetDlyLine, 6-70
Sinh, 12-45
sliding window technique, 13-54, 13-66
SmoothedPowerSpectrumAurora, 8-216
SortAscend, 5-55
SortDescend, 5-55
special vector functions
 VectorJaehne, 4-37
 VectorRamp, 4-38
spectral data prequantization, 10-10
spectral values restoration, 10-14
speech codecs common functions
 AutoCorr, 9-14
 AutoCorr_NormE, 9-17
 AutoCorrLagMax, 9-15
 AutoScale, 9-12
 ConvPartial, 9-6
 CrossCorr, 9-18
 CrossCorrLagMax, 9-19
 DotProdAutoScale, 9-13
 InterpolateC_NR, 9-7
 InvSqrt, 9-14
 Mul_NR, 9-8
 MulC_NR, 9-9
 MulPowerC_NR, 9-11
 SynthesisFilter_G723, 9-20
speech, synthesized, 9-68
SplitC, 11-24
SplitScaled, 5-61
SplitVQ, 8-200
Spread, 10-47
Sqr, 5-39
Sqrt, 5-41, 12-8

-
- StaticInit(). See ippStaticInit
 - StaticInitCpu(). See ippStaticInitCpu
 - Statistical functions, 5-115 thru 5-138
 - DotProd, 5-136
 - Max, 5-117
 - MaxAbs, 5-119
 - MaxAbsIndx, 5-120
 - MaxEvery, 5-139
 - MaxIndx, 5-118
 - Mean, 5-127
 - Min, 5-120
 - MinAbs, 5-123
 - MinAbsIndx, 5-124
 - MinEvery, 5-139
 - MinIndx, 5-122
 - MinMax, 5-125
 - MinMaxIndx, 5-126
 - Norm, 5-131
 - Normalize, 5-53
 - NormDiff, 5-133
 - Phase, complex, 5-68
 - PowerSpectr, complex, 5-69
 - StdDev, 5-129
 - Sum, 5-115
 - StdDev, 5-129
 - StepSizeUpdateAECNLMS, 8-268
 - Streaming SIMD Extensions, 1-1
 - string functions
 - Compare, 11-11
 - CompareIgnoreCase, 11-13
 - CompareIgnoreCaseLatin, 11-13
 - Concat, 11-22
 - ConcatC, 11-23
 - Equal, 11-14
 - Find, FindRev, 11-3
 - FindC, FindRevC, 11-5
 - FindCAny, FindRevCAny, 11-7
 - Hash, 11-21
 - Insert, 11-8
 - Lowercase, 11-19
 - LowercaseLatin, 11-19
 - RegExpFind, 11-31
 - RegExpFree, 11-27
 - RegExpGetSize, 11-29
 - RegExpInit, 11-28
 - RegExpSetMatchLimit, 11-30
 - Remove, 11-9
 - ReplaceC, 11-17
 - SplitC, 11-24, 11-26
 - TrimC, 11-15
 - TrimCAny, 11-16
 - TrimEndCAny, 11-16
 - TrimStartCAny, 11-16
 - Uppercase, 11-18
 - UppercaseLatin, 11-18
 - Sub, 5-30
 - SubC, 5-27
 - SubCRev, 5-28
 - SubRow, 8-13
 - Sum, 5-115
 - SumColumn, 8-9
 - SumColumnAbs, 8-180
 - SumColumnSqr, 8-181
 - SumLn, 5-51
 - SumMeanVar, 8-71
 - SumRow, 8-11
 - SumRowAbs, 8-182
 - SumRowSqr, 8-183
 - SVD, SVDSort, 8-184
 - SwapBytes, 5-56
 - SynthesisFilter_G728, 9-241
 - SynthesisDownFilter_SBR, 10-172
 - SynthesisDownFilterGetSize_SBR, 10-161
 - SynthesisDownFilterInit_SBR, 10-165
 - SynthesisFilter_AMRWB, 9-189
 - SynthesisFilter_G723, 9-20, 9-112
 - SynthesisFilter_G729, 9-68
 - SynthesisFilter_SBR, 10-169
 - SynthesisFilterGetSize_SBR, 10-160
 - SynthesisFilterGetStateSize_G728, 9-240
 - SynthesisFilterInit_G728, 9-240
 - SynthPQMF_MP3, 10-105
- T**
- TabsCalculation_Aurora, 8-221

Tan, 12-32
Tanh, 12-47
Threshold, 5-74
Threshold_GT, 5-77
Threshold_GTAbs, 5-80
Threshold_GTVal, 5-84
Threshold_LT, 5-77
Threshold_LTAbs, 5-80
Threshold_LTInv, 5-88
Threshold_LTVal, 5-84
Threshold_LTValGTVal, 5-84
tilt, 9-76
TiltCompensation_G723, 9-114
TiltCompensation_G729, 9-75
ToeplitzMatrix_G723, 9-106
ToeplitzMatrix_G729, 9-55
Tone, 4-14
Tone_Direct, 4-15
ToneFree, 4-10
tone-generating function, 4-14
tone-generating functions
 Tone_Direct, 4-15
 ToneFree, 4-10
 ToneGetStateSizeQ15, 4-11
 ToneInitAllocQ15, 4-9
 ToneInitQ15, 4-12
 ToneQ15, 4-13
ToneGetStateSizeQ15, 4-11
ToneInitAllocQ15, 4-9
ToneInitQ15, 4-12
ToneQ15, 4-13
transcendental mathematical functions, 12-1
Transform support functions, 7-6 thru 7-17
Triangle, 4-23
Triangle_Direct, 4-25
TriangleFree, 4-19
triangle-generating function, 4-23
triangle-generating functions
 Triangle_Direct, 4-25
 TriangleFree, 4-19

TriangleGetStateSizeQ15, 4-20
TriangleInitAllocQ15, 4-18
TriangleInitQ15, 4-21
TriangleQ15, 4-22
TriangleGetStateSizeQ15, 4-20
TriangleInitAllocQ15, 4-18
TriangleInitQ15, 4-21
TriangleQ15, 4-22
TrimC, 11-15
TrimCAny, 11-16
TrimEndCAny, 11-16
TrimStartCAny, 11-16

U

Understanding windowing functions, 5-102 thru 5-103
uniform distribution functions
 RandUniformFree, 4-27
 RandUniformGetSize, 4-29
 RandUniformInit, 4-28
 RandUniformInitAlloc, 4-26
 RandUnifrom, 4-29
 RandUnifrom_Direct, 4-30
UnitCurve, 8-40
Unpack of packed data, 7-6 thru 7-12
 ConjCcs, 7-11
 ConjPack, 7-7
 ConjPerm, 7-9
UnpackADIFHeader_AAC, 10-119
UnpackADTSFrameHeader_AAC, 10-120
UnpackFrameHeader_MP3, 10-95
UnpackScaleFactors_MP3, 10-97
UnpackSideInfo_MP3, 10-96
UpdateGConst, 8-169
UpdateLinear, 6-10
UpdateMean, 8-166
UpdateNoisePSDMCRA, 8-251
UpdatePathMetricsDV, 5-101
UpdatePower, 6-11
UpdateVar, 8-167
UpdateWeight, 8-168

Uppercase, 11-18
 UppercaseLatin, 11-18
 User filter banks wavelet transforms, 7-82 thru 7-97

V

VAD_AMRWB, 9-191
 VAD1_GSMAMR, 9-161
 VAD2_GSMAMR, 9-164
 VADDecision_Aurora, 8-235
 VADFlush_Aurora, 8-236
 VADGetBufSize_Aurora, 8-234
 VADGetSize_AMRWB, 9-190
 VADInit_AMRWB, 9-191
 VADInit_Aurora, 8-235
 VarColumn, 8-143
 Variable Length coding, 13-4
 variable length coding table, 13-4
 VecMatMul, 8-19
 Vector correlation functions
 AutoCorr, 6-5
 vector initialization functions
 Copy, 4-3
 Move, 4-5
 Set, 4-6
 Zero, 4-7
 Vector quantization functions, 8-187 thru 8-203
 CdbkFree, 8-195
 CdbkGetSize, 8-190
 CdbkInit, 8-191
 CdbkInitAlloc, 8-193
 FormVector, 8-187
 FormVectorVQ, 8-201
 GetCdbkSize, 8-195
 GetCodebook, 8-196
 SplitVQ, 8-200
 VQ, 8-197
 vector quantizer, 9-38
 VectorJaehne, 4-37
 VectorRamp, 4-38
 version information function, 3-2
 Viterbi decoder functions, 5-99 thru 5-102

BuildSymbITableDV4D, 5-101
 CalcStatesDV, 5-100
 GetVarPointDV, 5-99
 UpdatePathMetricsDV, 5-101
 VLC functions, 10-37
 VLCCountBits, 13-10
 VLCCountEscBits_AAC, 10-43
 VLCCountEscBits_MP3, 10-42
 VLCDecodeBlock, 13-15
 VLCDecodeEscBlock_AAC, 10-40
 VLCDecodeEscBlock_MP3, 10-37
 VLCDecodeFree, 13-12
 VLCDecodeGetSize, 13-14
 VLCDecodeInit, 13-12
 VLCDecodeInitAlloc, 13-11
 VLCDecodeOne, 13-16
 VLCEncodeBlock, 13-8
 VLCEncodeEscBlock_AAC, 10-46
 VLCEncodeEscBlock_MP3, 10-44
 VLCEncodeFree, 13-6
 VLCEncodeGetSize, 13-7
 VLCEncodeInit, 13-6
 VLCEncodeInitAlloc, 13-5
 VLCEncodeOne, 13-9
 Voice Activity Detection functions
 FindPeaks, 8-274
 Periodicity, 8-276
 PeriodicityLSPE, 8-275
 VQCodeBookFree, 10-50
 VQCodeBookGetSize, 10-50
 VQCodeBookInit, 10-49
 VQCodeBookInitAlloc, 10-48
 VQMainSelect, 10-53
 VQReconstruction, 10-57
 VQSingle_Sort, 8-198
 VQSingle_Thresh, 8-198

W

Wavelet transform functions, 7-74 thru 7-97
 WTFwd, 7-86

WTFwdFree, 7-85
WTFwdGetDlyLine, 7-89
WTFwdInitAlloc, 7-82
WTFwdSetDlyLine, 7-89
WTHaarFwd, 7-77
WTHaarInv, 7-77
WTInv, 7-92
WTInvFree, 7-85
WTInvGetDlyLine, 7-95
WTInvInitAlloc, 7-82
WTInvSetDlyLine, 7-95
WaveProcessing_Aurora, 8-223
WeightedMeanColumn, 8-146
WeightedMeanVarColumn, 8-149
WeightedSum, 8-186
WeightedVarColumn, 8-148
weighting matrix, 9-99
WeightingFilter_GSMFR, 9-224
WienerFilterDesign_Aurora, 8-218
WinBartlett, 5-104
WinBlackman, 5-105
Windowing functions, 5-102 thru 5-115
 WinBartlett, 5-104
 WinBlackman, 5-105
 WinHamming, 5-109
 WinHann, 5-111
 WinKaiser, 5-113
WinHamming, 5-109
WinHann, 5-111
WinHybrid_G728, 9-254
WinHybrid_G729E, 9-87
WinHybridGetStateSize_G728, 9-252
WinHybridGetStateSize_G729E, 9-86
WinHybridInit_G728, 9-253
WinHybridInit_G729E, 9-86
WinKaiser, 5-113
WTFwd, 7-86
WTFwdFree, 7-85
WTFwdGetDlyLine, 7-89
WTFwdInitAlloc, 7-82
WTFwdSetDlyLine, 7-89

WTHaarFwd, 7-77
WTHaarInv, 7-77
WTInv, 7-92
WTInvFree, 7-85
WTInvGetDlyLine, 7-95
WTInvInitAlloc, 7-82
WTInvSetDlyLine, 7-95

X

Xor, 5-11
XorC, 5-10

Z

Zero, 4-7
ZeroMean, 8-22
ZLIB, 13-45
ZLIB functions, 13-45 thru 13-91
 Adler32, 13-75
 CRC32, 13-76
 DecodeLZ77, 13-66
 DecodeLZ77DynamicHuff, 13-69
 DecodeLZ77FixedHuff, 13-68
 DecodeLZ77GetBlockType, 13-67
 DecodeLZ77GetPairs, 13-72
 DecodeLZ77GetSize, 13-64
 DecodeLZ77GetStatus, 13-73
 DecodeLZ77Init, 13-64
 DecodeLZ77InitAlloc, 13-65
 DecodeLZ77Reset, 13-75
 DecodeLZ77SetPairs, 13-73
 DecodeLZ77SetStatus, 13-74
 DecodeLZ77StoredBlock, 13-71
 EncodeLZ77, 13-53
 EncodeLZ77DynamicHuff, 13-57
 EncodeLZ77FixedHuf, 13-55
 EncodeLZ77Flush, 13-59
 EncodeLZ77Free, 13-53
 EncodeLZ77GetPairs, 13-60
 EncodeLZ77GetSize, 13-51
 EncodeLZ77GetStatus, 13-61
 EncodeLZ77Init, 13-50
 EncodeLZ77InitAlloc, 13-52

EncodeLZ77Reset, 13-63
EncodeLZ77SelectHuffMode, 13-54
EncodeLZ77SetPairs, 13-61
EncodeLZ77SetStatus, 13-62
EncodeLZ77StoredBlock, 13-58